摘要

缓冲区溢出漏洞一直是安全漏洞的最常见的一种形式。近年来，在 CERT/CC（Computer Emergency Response Team/ Coordination Center）发布的公告中，关于缓冲区溢出漏洞的占 50%以上。缓冲区溢出问题主要出现在 C/C++这类老牌的编程语言中，而在新一代的编程语言，例如 Java、C#中不存在。一个最重要的原因就是 C/C++允许通过指针进行间接内存访问但没有缓冲区边界检查和提供了大量对缓冲区可能存在不安全操作的库函数，在 Windows 操作系统中也存在类似的函数。因此，如果能采用有效的手段对缓冲区溢出漏洞进行检测，将能极大地提高软件系统的安全性。

从一个程序中检测出可能的安全漏洞是一件非常困难和耗时的事情。目前针对缓冲区溢出漏洞的检测主要分为两类：静态检测的方法和动态检测的方法。通过使用静态和动态分析工具，可以从一定程度上减轻漏洞检测的困难。

本文研究分析了与缓冲区溢出相关的基本原理并分析了国内外缓冲区溢出检测的典型技术和工具。通过分析目前几个开源的源代码扫描工具，以及参与的一个源代码静态扫描工具的开发，本文提出一种对二进制代码的静态检测方法：对二进制代码的反汇编代码进行操作语义的分析，提取与缓冲区溢出相关的语法成分，然后以 C 库和 Windows 操作系统中不安全函数为出发点进行缓冲区溢出的分析，并基于此方法实现了一个原型系统。实验证明，该原型系统能够检测出一些栈缓冲溢出的情况。但由于完全实现的工作量太大，因此没有考虑复杂的数据结构和控制流等情况。

本文提出的静态检测方法是基于对二进制代码反汇编后的代码，实质上也可以认为是一种基于源代码的静态检测技术。这方面的研究是对主流技术的一种重要补充，在得不到源代码的情况下，可以作为一种备用手段使用。

关键词：缓冲区溢出，栈溢出，静态分析，二进制代码

Abstract

Buffer overflow has been the most common form of security vulnerability. The number of buffer overflows account for about 50% above of all the vulnerabilities according to the CERT/CC (Computer Emergency Response Team/ Coordination Center) statistic at recent years. The buffer overflow problem is typical of old programming languages, such as C and C++, whilst it cannot arise in the new generation of languages, e.g. Java and C#. A most important reason is that the C and C++ languages allow indirect access memory location by pointer without boundary check and there are many unsafe functions in library which may cause buffer overflow. And the Windows operating system also has some similar unsafe functions. No doubt, if can adopt effective means to detect this kind of security vulnerabilities, it will greatly enhance the security of the software systems.

Detecting possible buffer overflows in a program is a difficult and time consuming task. Methods that detect buffer overflows are generally divided into two kinds at present: dynamic and static method. The difficult of detecting can be alleviated a certain extent by using static and dynamic software analysis tools.

This thesis presents research focused on the fundamental issues surrounding the buffer overflow vulnerability and some typical methods and tools used on buffer overflow detecting. On the basis of analyzing a few of open source tools of source code based and taking part in a source code based scanning tool's developing, a new buffer overflow static detecting method is proposed. That is, analyze the operate semantics of disassembled binary code, pick up the information that buffer overflow concerned by syntactic analysis, and then proceed our analysis from the set of so-called "dangerous functions". On basis of this, we implement a prototype system that can locate certain types of buffer overflow vulnerabilities by experiments. Considering the heavy work load of complete implementation, we take no account of complicated data structure, control flow etc.

The static detecting method proposed by this thesis is base on the disassembled binary code, so it also can be considered to be a source code based static detecting

method in fact. But the method proposed by this thesis is performed on a released program, a different approach compared to the many previous studies that focus on static source code analysis, that it can be a means in support of the main detecting methods.

Keywords: buffer overflow, stack overflow, static analysis, binary code

图目录

表目录

独 创 性 声 明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

签名：向文杰　日期：2007 年 4 月 24 日

关于论文使用授权的说明

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

签名：向文杰　导师签名：李敦

日期：2007 年 4 月 24 日

第一章 引言

1.1 研究背景

利用漏洞发起攻击是互联网最大的安全隐患。根据美国 CERT/CC（Computer Emergency Response Team/ Coordination Center）统计，自 1995 年到 2006 年漏洞累计达到 30780 个，2006 年共报告漏洞 8064 个，平均每天超过 22 个。在过去的十年中，以缓冲区溢出为类型的安全漏洞是最为常见的一种形式[1]。更为严重的是，缓冲区溢出漏洞占了远程网络攻击的绝大多数，这种攻击可以使得一个匿名的 Internet 用户有机会获得一台主机的部分或全部的控制权！由于这类攻击使任何人都有可能取得主机的控制权，所以它代表了一类极其严重的安全威胁。缓冲区溢出攻击之所以成为一种常见安全攻击手段，其原因在于缓冲区溢出漏洞太普通了，并且易于实现。而且，缓冲区溢出成为远程攻击的主要手段，其原因在于缓冲区溢出漏洞给予了攻击者他所想要的一切：植入并且执行攻击代码。被植入的攻击代码以一定的权限运行有缓冲区溢出漏洞的程序，从而得到被攻击主机的控制权。

在 1998 年 Lincoln 实验室用来评估入侵检测的 5 种远程攻击中，有 3 种是基于社会工程学的信任关系，2 种是缓冲区溢出。而在 1998 年 CERT 的 13 份建议中，有 9 份是与缓冲区溢出有关的，在 1999 年，至少有半数的建议是和缓冲区溢出有关的，在 2003 年，28 份建议中和缓冲区溢出有关的有 12 份[2]。国内绿盟科技记录的 1895 条有关漏洞中，缓冲区溢出漏洞占 410 条，几乎涉及到所有的操作系统和应用程序。在 Bugtraq 的调查中，有 2/3 的被调查者认为缓冲区溢出漏洞是一个很严重的安全问题。

绿盟科技[3]根据安全漏洞的严重程度、影响范围等因素综合评出 2006 年度的十大安全漏洞，缓冲区溢出漏洞占 4 个，分别包括：Microsoft Windows Server 服务远程缓冲区溢出漏洞（MS06-040），Microsoft Windows MSDTC 堆溢出漏洞（MS06-018），Microsoft Windows Workstation 服务 NetpManageIOCConnect 远程栈溢出漏洞（MS06-070）和 Apache mod_rewrite 模块单字节缓冲区溢出漏洞。绿盟科技在 2006 年共发布 CVE 漏洞 8 个，而缓冲区溢出漏洞就占了 5 个。其中，Microsoft IE6 urlmon.dll 长 URL 缓冲区溢出漏洞，源于一个 lstrncpyA 函数的错误调用，可能造成一个堆数据结构的破坏，通过精心构造数据可能导致执行任意代

码。winamp m3u 列表文件处理缓冲区溢出漏洞，winamp 可以通过加载.m3u 列表文件来播放其中的文件，当某个文件播放停止时，winamp 会重新设置程序标题。这时 winamp 错误的使用了 strncpy 函数，导致一个静态缓冲区溢出。

因此，如果能采用有效的手段对缓冲区溢出漏洞进行检测，对提高软件本身的健壮性和安全性具有十分重要的意义。

1.2 本文主要工作

本文主要工作为：在研究利用缓冲区溢出进行的主要攻击手段，分析已有缓冲区溢出检测工具，以及参加和四川省安全厅合作的针对源代码的栈缓冲区溢出检测工具的开发的基础上，给出了一种针对二进制代码的栈缓冲溢出静态检测的方法并用语法分析和操作语义的方法加以严格描述，并最终实现一个针对二进制代码的栈缓冲溢出自动检测原型系统。具体包括：

1. 二进制代码静态扫描的特征研究

基于二进制代码的静态扫描，相对较 C/C++的源代码扫描复杂。二进制代码文件的精确格式分析、汇编级的指令还原、汇编指令级的漏洞语法研究都是基于二进制代码的静态扫描所特有的。研究以下二进制代码相关特征是开发的基础：

二进制代码文件的格式特征及利用；

编译器版本特征研究及利用；

敏感数据的编译器优化特征研究及利用。

2. 缓冲区溢出漏洞二进制代码级模式研究

结合二进制代码静态扫描的特征，发现漏洞特征码和漏洞的语法模式，并将基于 C/C++源码的静态分析技术向汇编级的静态分析进行移植和扩展是整个问题的关键。具体内容包括：

特征码研究；

控制流提取及数据流跟踪研究；

语法级模式。

3. 漏洞发现的词法、语法分析器生成

通过漏洞模式的构建和特征码的整理，设计一个结构合理、性能良好的语法分析引擎。具体内容包括：

词法、语法单元的粒度设计；

控制流的再现；

漏洞模式的匹配和后继语义分析。

通过实例测试证明，该原型系统能有效的检测出二进制代码中的一些栈缓冲溢出的情况。

1.3 论文结构

本文第一章为引言，主要介绍课题的一些背景情况和研究内容。

第二章介绍缓冲区溢出的原理，并比较了目前流行的缓冲区溢出检测方法和工具，提出了本文的栈缓冲区溢出检测方法。

第三章介绍本文的基础研究，包括静态分析基础和词法、语法分析的相关理论基础。静态分析基础将介绍如何从反汇编代码中识别与缓冲区溢出检测相关的语法元素；词法、语法分析是本文的栈溢出检测方法的核心，因此该章将详细介绍与本文相关的词法、语法分析理论，并介绍了本课题使用的两个开发工具：词法分析程序和语法分析程序的自动生成器。

第四章详细介绍对二进制代码的静态扫描原型系统的设计与实现。

第五章介绍对实现的二进制扫描原型系统进行的测试和分析。

最后是对本文工作的总结。

第二章　缓冲区溢出原理及检测技术

缓冲区溢出是一个众所周知的安全问题。缓冲区溢出攻击仍然是现在最有效的软件攻击方法之一，并且在接下来的许多年里，情况仍然会是这样的。缓冲区溢出攻击可导致程序运行失败、系统当机、重新启动等后果。更为严重的是可以利用它执行非授权的指令，获得被攻击主机的部分或者全部特权，进而进行各种非法操作。因此，针对缓冲区溢出漏洞检测方法的研究具有十分重要的意义。本章简单介绍了缓冲区溢出的原理，并比较了目前流行的缓冲区溢出检测方法和工具，然后提出了本文的栈缓冲区溢出检测方法。

2.1 缓冲区溢出原理简介

所谓缓冲区，简单说来就是程序运行时内存中的一块连续的区域。例如 C 语言中经常要用到的数组，其中最常见的是字符数组。在一个程序中，会声明各种变量。静态全局变量是位于数据段并且在程序开始运行的时候被加载。而程序的动态的局部变量则分配在堆栈里面。如果向一个缓冲区复制数据，但是复制的数据量又比缓冲区大的时候，就会发生缓冲区溢出[28]。

缓冲区溢出主要出现在 C 和 C++语言中，因为这些语言不执行数组边界检查和类型安全检查。C/C++开发人员创建非常接近硬件环境运行的程序，允许直接访问内存和寄存器。这样的程序可以获得优异的性能，但也带来了严重的安全问题。

标准 C 库中还存在了像 strcpy、strcat、sprintf 等不安全的字符串操作函数。Windows 操作系统中也存在 lstrcpy、lstrcat 等类似的函数。不正确的使用这些函数也是导致缓冲区溢出的重要原因。

缓冲区溢出问题又常常被归结为通道交错问题。编译器将数据和控制管理信息混合存放。例如，函数的缓冲区位于堆栈中，同时该函数的返回地址就在缓冲区后。又如，动态内存分配的块管理信息和数据连续存放。类似的管理信息还包括异常处理程序地址、函数指针等等。典型的格式化字符串漏洞也属于这种情况。数据通道，只被复制而不被解释。控制通道，具有控制功能，需要系统解释。如果程序正常执行，这两种通道可以工作的很好，不会出现问题。但是，如果这两个不同的通道交错在一起时，漏洞就会暴露出来。攻击者利用对数据通道的写权

限改写了控制通道的内容，那么系统就有可能出现控制上的紊乱，而导致攻击。

因此，可以归纳出缓冲区溢出的必要条件有：

1. 使用非类型安全的语言，如 C/C++；
2. 以不安全的方式访问或复制缓冲区；
3. 编译器将缓冲区放在内存中关键数据结构相邻的位置。

缓冲区溢出攻击技术的种类很多。从攻击原理分，缓冲区溢出攻击可以分为栈溢出、普通堆溢出、高级堆溢出和格式化字符串溢出等；从攻击方式分，又可以分为本地溢出和远程溢出。一个标准的缓冲区溢出攻击大致可以分为三个步骤：

1. 在目标系统中注入攻击代码；
2. 发现目标系统运行过程中可以控制的缓冲区漏洞及其触发条件；
3. 精心构造满足溢出条件的超长字符串，使得溢出发生时可以截获目标系统的控制权，指向攻击代码、或者指向更高级别的系统进程配合攻击代码完成恶意的攻击。

2.2 缓冲区溢出检测技术

当前对于缓冲区溢出的检测方法总体上分为两类：静态检测和动态检测的方法。

静态检测又可分为基于源代码的静态检测和基于二进制代码的静态检测。基于源代码的静态检测技术是目前研究相对较多的一个分支，其特征是通过对源代码的扫描和分析，对缓冲区溢出发生的模式进行识别，从而完成溢出漏洞的检测。基于二进制代码的静态检测技术的研究仍然比较少见，目前典型的研究是通过一些反汇编工具对目标代码进行处理，然后再依赖一些源代码静态检测技术进行处理。这也正是本文所采用的方法。

动态检测技术也可分为基于源代码的动态检测和基于二进制代码的动态检测。基于源代码的动态检测通过在执行时对程序内存中的访问情形加以控制来完成溢出漏洞的检测。比如通过对在源码中插入一些约束和判断的模块，然后在编译后的程序运行期间对有关变量和堆栈区域的监控来检测漏洞。这种检测技术改变了栈的结构，需要重新编译源代码。基于二进制代码的动态检测通过自动化工具生成测试数据，通过仿真攻击状态下应用程序的执行状态来判断缓冲区溢出漏洞位置，又被称为黑箱分析。这种通过运行发现缓冲区溢出的动态检测方法非常准确，而且不需要源代码，但是分析的代价大，效率不高。

2.2.1 静态检测技术

最简单的静态检测技术是使用与 UNIX 平台下的 grep 类似的工具，搜索源代码中可能存在的一些不安全的库函数的调用。

ITS4、RATS 和 Flawfinder 是目前发展比较成熟的源码扫描工具，其基本原理是对源程序进行词法分析，得到 token 序列，然后与其维护的漏洞数据库中的信息比较，发现可能的缺陷，给出提示。这类检测技术实现简单，算法效率高，能够较全面的覆盖系统代码，但是由于没有考虑到语法和语义层次的信息，所以这些检测工具能够检测出系统的大量漏洞，但很多是误报。而且产生的结果集很大，人工审计仍然很困难[5][14][15][16]。

Flawfinder[14]工具是一个 Python 程序，可以用来检查 C 和 C++程序。Flawfinder 的速度非常快，在一般的台式机上处理几千行的 C 程序只需要几秒钟。在扫描漏洞的过程中，Flawfinder 也具有一定的智能性。例如在对于一个特别编写的不安全的 C 程序进行扫描的过程中，Flawfinder 可以区分出 strcpy()对于固定长度的字符处理和变长的字符处理，从而区分出真正的漏洞和错误的报警。更进一步，Flawfinder 还能够识别字符串在国际化中的使用。

RATS（the Rough Auditing Tool for Security）[15]也是一个影响比较大的源代码扫描工具，目前正处于比较活跃的开发过程中，可以检查使用 C，C++，Perl，PHP 和 Python 等语言编写的源程序。RATS 可以输出 XML 格式的错误信息，因此在使用 RATS 之前，还需要安装 XML 处理工具 Expat[17]。在 RATS 的运行过程中，用户可以通过配置来改变其输出的错误信息级别，缺省是中级。可选的漏洞知识库和一致的输出函数可以接受用户的自定义输入，因此更方便用户使用自己的专门数据来发现特定的错误。目前 RATS 中已知的缺点包括它所使用的贪婪匹配算法，使用这种算法，printf 能够匹配 print，vsnprintf 以及其它许多类似的函数，这就使得过滤错误的命中变得比较困难。

ITS4（the Software Stupid Source Scanner）[16]是基于 Linux 和 UNIX 的扫描工具，可以扫描 C 和 C++语言所编写的程序，用于发现一些最普遍的安全性相关的漏洞。由于许多 ITS4 的开发人员转向开发 RATS 和 Flawfinder，因此它的开发趋于平和。但某些吸引人的特点在 RATS 或者 Flawfinder 中得到了发展，例如行忽略技术、跟踪用户输入以及可选的漏洞知识库等。ITS4 可与 UNIX 下的编辑工具 Emacs 结合起来运行，在程序员编码的过程中同步完成检查，可以随时的提示程序员。这也是 ITS4 与其它源码扫描工具相比所独有的一个特点。

David Wagner[11]将缓冲区溢出的检测问题转化为整数区间分析的形式进行处理。在该方法中，他们提出了两个新观点：将 C 字符串看作抽象数据类型；将缓冲区视为整数范围对，包括分配范围和使用范围。将分配给字符串变量的缓冲区长度定义为 alloc(s)，字符串变量当前使用的缓冲区长度定义为 len(s)，s 为系统中定义的字符串常量。如果程序当前使用的缓冲区长度 len(s)大于系统为其分配的缓冲区长度 alloc(s)，那么就有可能存在缓冲区溢出漏洞。在考虑缓冲区长度的同时，也必须注意到缓冲区的位置，也就是与变量相关的缓冲区范围的长度与位置。这样，对缓冲区溢出漏洞的发掘也就成为了对整数范围的追踪问题。整个工作分为两部分，其一是对字符串操作产生相应的整数范围限制；其二是对产生的限制进行快速而准确的分析，得到最后的漏洞报告信息。为此，他们使用图论技术构建了一个有效的解决整数范围限制的算法。由于不精确的范围分析，该方法最大的局限性就是过多的误报。

Vinod Ganapathy[18]等人基于线性规划和静态分析理论提出了一种轻量级别的检测算法。该算法首先利用工具软件生成一些指针相关的信息和抽象语法树，然后基于此生成一些线性约束，然后利用线性规划的思想来完成区间分析，检测可能出现的溢出漏洞。

Lint 是由 Andy Lester 发布的一个开源工具，能够检查可能的空指针、在释放内存之后使用了该指针、赋值次序问题以及拼写错误等。一个 C/C++编译器通常假设程序是正确的，而 Lint 恰恰相反，因此，它优于编译器执行的一般性检查。Lint 还可以贯穿多个文件来执行它的错误检查和代码分析。目前有两个流行的 Lint 工具：PC-lint 和 Splint（原 LCLint）。PC-lint 是一个由 Gimpel Software 提供的支持 C/C++的商用程序，其中的内容非常广泛，只是选项就有 300 多个，涉及到程序编译及语法使用中的方方面面。Splint 是在 Lint 工具的基础上改造而成的，专门针对程序的安全问题。它要求手工加入注释，然后根据这些注释进行语法分析，利用了语法树[26][27]。

静态检测主要是通过对源代码扫描来判断是否存在缓冲区溢出的漏洞，其优点是在软件发布之前就修正所发现的弱点，并且不会造成程序执行的负担。国内也有许多学者和研究机构进行了这方面的研究，如参考文献[43][44][45][46][47]。

2.2.2 动态检测技术

目前，动态检测研究的主要方面是数组边界检查和如何保证返回指针的完整性。典型的方法包括基于编译器运行时的边界检查和基于库运行时的边界检查。

StackGuard[19]是一种通过程序指针完整性检测来防范堆栈溢出问题的编译器技术。其原理是基于栈的缓冲溢出攻击时，栈中溢出的缓冲区和返回地址之间的值也都被修改。于是在栈缓冲区和返回地址之间放一个“canary”值，当函数返回时，若发现这个值被修改，那么就检测到了溢出攻击。StackGuard 作为 gcc 的一个补丁，修改了函数建立和销毁部分的代码，由这些代码来完成“canary”值的插入和检查工作。为了使这种保护机制有效，就不能给攻击者在攻击字符串中夹杂伪造“canary”值的机会，所以这个值可以在程序执行时随机产生，攻击者就不能通过搜索程序的二进制文件得到“canary”值。这种方法的主要缺陷是存在拒绝服务攻击，因为当检测到攻击时，终止了程序的执行。另外，该方法只保护返回地址，没有保护基指针，函数指针。

StackShield[20]也是 gcc 编译器的一个补丁，提供三种保护方式，其中两种保护返回地址，一种保护函数指针。返回地址保护所采用的方法有：其一是创建另一个堆栈用来存储函数返回地址的一份拷贝。在受保护的函数的开头和结尾分别增加一段代码，当函数返回地址压栈时，同时把它压入新创建的栈中，当函数返回时，用这个栈的栈顶值作为返回地址。这样，即使函数返回地址被修改，也能保证程序能够继续的正确执行下去。二是设一个全局变量，当函数调用时，把返回地址存入这个值中，当函数返回时，比较这个值和栈中返回地址的值，如果不同就检测到了攻击。函数指针的保护方法是限制可执行代码只能出现在代码段中。堆、栈和 BSS 段都不能有函数指针所指向的可执行的代码。当函数调用时，进行边界检查，如果所指向的区域地址不在代码段之内，终止程序的执行。这种方法的主要缺陷是拒绝服务攻击仍然可能存在，而且代码都要保存在代码段。

RAD（Return Address Defender）[8]与 StackShield 非常类似，也用一个额外的栈，每当函数调用时压栈，当函数返回时，比较返回地址和这个额外的栈的栈顶所记录的值是否相同，如果相同，继续执行，否则，终止执行，给出警告。主要缺陷是拒绝服务攻击仍然可能，而且这个额外的栈本身也要被保护，增加了开销。

IBM 的 stack-smashing 保护程序 SSP（又名 ProPolice）[23]是 StackGuard 的方法的一种变化形式。像 StackGuard 一样，SSP 使用一个修改过的编译器在栈缓冲区和返回地址之间插入一个“canary”值以检测堆栈溢出。但是，它给这种基本的思路添加了一些变化：改变栈的分配结构，不管源程序中各个变量的声明顺序，把它们都放在低端地址，把返回地址和基指针放在高端地址，这样即使溢出也不会修改到高端地址处存放的返回地址。默认情况下，它不会检测所有的函数，而只是检测确实需要保护的函数。从理论上将，这样会稍微削弱保护能力，但这种

默认行为改进了性能，同时仍然能够防止大多数问题。

Kyung-suk Lhee[24]等人提出了一种针对 C 的库函数调用产生的溢出问题的动态检测技术。他们对 GNU C 的编译器进行了扩展，使得编译后的程序在内存中维持有局部变量、函数参数、全局变量等类型的缓冲区的类型信息，并对内存堆中分配的缓冲区大小进行了记录。依赖这些信息，程序在运行期间可以对某些溢出情形作出判断。

此外，还有一些动态检测方法，如 libsafe、libverify[25]等对 C 库中的不安全函数做了重新实现。这些动态检测方法都仔细研究了编译中对变量声明，函数调用的栈、堆的分配，着眼于保护已分配的返回地址和函数指针不被修改。

2.3 本文的栈溢出检测方法

目前比较成型的静态分析工具所采用的方法有很多缺点：都只做到了词法分析，然后根据构造的数据库，通过匹配的方法判断是否符合溢出模式；不够准确，很多错误检测不到，或者检测到的并非错误；唯一做了语法分析的工具是手工在源代码中加入注释，并不是一个全自动的检测工具。

与动态分析技术相比，静态分析技术无需庞大的测试数据，无需精心设计测试用例，无需过多的程序运行负担是它相对于动态分析的最大优点；对漏洞很少有漏报的情况，而且设计相对简单，运行开销也相对较小。通过结合高级或低级语言自身的语法特征进行较高程度的漏洞语法模式定义，做进一步较精确的语法分析，误报率将会大大减小。

对二进制代码的漏洞发现，同样可以通过静态分析实现：经一定的结构分析和提取后，转化为汇编级源码，然后对汇编指令流进行静态分析。目前，针对汇编一级的静态扫描研究还很少。汇编级的程序流相对 C/C++源程序而言，其语法分析较困难，特别是对缓冲溢出相关的数组操作，经不同的编译器产生的代码区别很大；另外，汇编指令一级的语法分析需考虑的问题范畴更加广泛、复杂。但是，由于汇编指令流中存在和漏洞相对应的模式，而且相对 C/C++源码的分析，汇编指令一级的静态分析也有其优点：避免了 C/C++中多线程等导致的程序控制流程烦杂不易分析的特点，从而分析工作较直观、简单。所以，基于静态分析的汇编级漏洞发现是可行的。

通过分析目前几个开源的源代码扫描工具，以及参与的一个源代码静态扫描工具的开发，本文提出一种对二进制代码的栈缓冲溢出静态检测方法。该方法借鉴

了参与开发的源代码静态扫描工具中的相关技术，其技术路线可描述如下：

1. 反汇编二进制文件，分析与栈缓冲溢出漏洞相关的特征和语法模式；
2. 将基于 C/C++源码的静态分析技术向汇编级的静态分析进行移植和扩展；
3. 设计一个结构合理、性能良好的语法分析引擎进行与栈缓冲溢出相关语法成分的分析；
4. 以可能导致栈缓冲溢出的 C 库函数和 Windows 操作系统中类似的函数为出发点，进行是否可能导致栈缓冲溢出的分析。

我们的分析是基于反汇编后的代码，实质上也可以认为是一种基于源代码的静态检测技术。这方面的研究也是对主流技术的一种重要补充，在得不到源代码的情况下，可以作为一种备用手段使用。

2.4 有缓冲区溢出问题的 API

本文所实现的栈缓冲溢出检测工具的基本出发点是针对可能导致缓冲区溢出的库函数进行检查的方式来确定程序中是否存在安全隐患。因此，该小节列举出 C 运行库和 Windows 操作系统中可能导致缓冲区溢出的不安全函数。

1. strcpy、wcscpy、lstrcpy、_tcscpy 和_mbscpy
2. strcat、wcscat、lstrcat、_tcscat 和_mbscat

这些函数不检查目标缓冲区的长度，也不检查 null 或其它无效的指针。如果源缓冲区不是以 null 结尾的，那么结果是无法判断的。

3. strncpy、wcsncpy、lstrcpyn、_tcsncpy 和_mbsnbcpy

不能保证这些函数会用 null 来结束目标缓冲区，它们也不检查 null 或其它无效的指针。

4. strncat、wcsncat、_tcsncat 和_mbsnbcat

这些函数检查被拷贝的字符数是否与缓冲区中的字符数是相同的，而不是检查缓冲区的大小。这些函数依赖于源缓冲区和目标缓冲区是否以 null 结尾。

5. memcpy 和 CopyMemory

目标缓冲区必须足够大，以便能够存储在长度参数中指定的字节数，否则会发生缓冲区溢出。

6. sprintf 和 swprintf

不能保证这些函数会用 null 结束目标缓冲区。除非严格的定义了字段的宽度，否则这些函数很难安全的使用。

7. gets

gets 函数是有害的，用这个函数就不可能写出安全的应用程序，因为它不检查被拷贝的缓冲区大小。

8. scanf，_tscanf，wscanf

像 gets 一样，scanf，_tscanf 和 wscanf 在使用%s 格式化字符的时候，由于%s 是没有边界的，因此都很难正确使用。

9. MultiByteToWideChar

这个函数的最后一个参数是字符串中宽字符的数目，而不是字节数。如果传递的是字节数，那么给出的缓冲区长度是实际长度的两倍。

2.5 小结

本章简单介绍了缓冲区溢出的原理，通过比较目前主流的缓冲区溢出检测技术，提出了一种对二进制代码的栈缓冲溢出检测方法。该方法通过对二进制代码的反汇编代码进行语法分析，识别出与栈缓冲溢出相关的语法成分并进行一定程度的语义分析，在此基础上实现一个栈缓冲溢出漏洞的检测工具。下一章将具体介绍如何从反汇编代码中识别与缓冲区溢出相关的语法成分，并结合相关的词法、语法分析理论，详细介绍了两个词法、语法分析器的自动生成工具：LEX（Lexical Analyzer Generator）和 YACC（Yet Another Compiler-Compiler），这也是本课题主要使用到的开发工具。

第三章　二进制代码的静态分析研究

识别二进制代码中与栈缓冲溢出相关的语法成分是本文所要实现的原型系统的关键。本章将通过比较源代码和相应二进制代码的反汇编代码，研究如何从二进制代码中识别高级语言的关键结构和与识别这些结构相关的词法、语法分析的理论基础。

3.1 静态分析基础

本文采用的栈缓冲溢出检测方法，首先就是对二进制代码进行反汇编。IDA（Interactive Disassembler）是一个强大的静态反汇编工具，本文其余的工作，都是基于IDA的反汇编结果进行的。因此，如何从二进制代码和反汇编代码中发现我们感兴趣的东西，是本节将要详细介绍的。

3.1.1　识别函数和库函数

IDA反汇编器能够分析call指令的操作数，这使它可以自动将函数分解成一系列的函数。除此之外，IDA还能够非常成功地应付绝大多数间接调用。不过，它尚不能处理复杂调用以及那些使用jmp指令的手工函数调用。

绝大多数非优化编译器会在函数的开头放入称为起始标志（prolog）的下列代码：

```
push    ebp
mov     ebp, esp
sub     esp, xx
```

一般而言，可以将起始标志的用途归结为：如果EBP寄存器用于对局部变量进行寻址（通常情况就是这样），那么在使用变量之前必须将它保存到堆栈中，否则，被调用函数将使父函数“崩溃”。接着，堆栈指针寄存器ESP的当前值要复制到EBP之中——这个操作称为打开堆栈页面，并且ESP的值会随着为局部变量分配的内存块大小而不断减少。

push ebp /mov ebp, esp /sub esp, xx序列可用于找到分析文件中所有函数，包括那些没有对它们进行直接引用的函数，特别是，IDA采用了这种技术。不过，优

化编译器知道如何像使用任何其它通用寄存器一样，通过 ESP 寄存器对局部变量进行寻址，以便将 EBP 做其它用途使用。在这种情况下，优化函数的起始标志仅仅含有一条 sub esp, xxx 指令，不幸的是，这个序列太短而不能当作辨认函数的标记。

另外，IDA 能识别大量的编译器标准库函数，这也是我们选择 IDA 这个反汇编器的原因。通过实际测试验证，IDA 能够识别 2.4 节列出的所有不安全的函数。

```
void fun()
{
    ......
}
int main()
{
    char buff1[200];
    char buff2[100];
    strcpy(buff2, buff1);
    fun();
    return 0;
}
```

```
.text:00401000 sub_401000   proc near
 ......
.text:00401004 sub_401000     endp
.text:00401010 _main        proc near
.text:00401010 var_130          = dword ptr -130h
.text:00401010 var_68           = dword ptr -68h
.text:00401010        push    ebp
.text:00401011        mov     ebp, esp
.text:00401013        sub     esp, 130h
.text:00401019        lea     eax, [ebp+var_130]
.text:0040101F        push    eax
.text:00401020        lea     ecx, [ebp+var_68]
.text:00401026        push    ecx
.text:00401027        call    _strcpy
.text:0040102C        add     esp, 8
.text:0040102F        call    sub_401000
.text:00401034        xor     eax, eax
.text:00401036        mov     esp, ebp
.text:00401038        pop     ebp
.text:00401039        retn
.text:00401039 _main        endp
```

图3-1 IDA 识别函数的例子

如图 3-1 所示源代码及其相应的二进制代码的反汇编代码，IDA 能够识别出普通函数和库函数 strcpy。对例子中的反汇编代码的分析可以发现，我们可以得到与源代码一样的函数及函数调用信息。从源代码中可以分析得到两个函数定义 main 和 fun，并且 main 函数内有两处函数调用，其中一处是普通函数调用 fun，另一处是不安全的函数调用 strcpy；从反汇编代码中也能得到两个函数定义 sub_401000 和_main，并且_main 有对 sub_401000 和_strcpy 的调用。

3.1.2 函数的参数

区分函数的参数是对程序的反汇编结果进行分析的关键步骤。给函数传递参数的方式有三种：堆栈方式、寄存器方式以及同时利用堆栈与寄存器来传递参数。参数可以按传值或者传递引用的方式进行传递。在第一种情况下，相应参数的拷贝被传递给函数。在第二种情况下，传递给函数的内容是指向参数的指针。

在 Win32 程序中，函数的调用主要有三种参数传递规范：C 规范（__cdecl）、Pascal 规范（PASCAL）和标准规范（__stdcall）。C 规范要求按照参数的声明顺序从右往左向堆栈传送参数，调用函数负责清除堆栈。Pascal 规范要求按照参数的声明顺序从左往右向堆栈传送参数，被调用函数负责清除堆栈。标准规范也要求按从右往左的顺序传送给堆栈，但清除堆栈的操作是由被调用函数执行的。

调用规范的类型可以大致由堆栈清除的方式来标识。如果它是由调用函数清除的，那么所处理的函数就是__cdecl 类型的，否则，所处理的函数就是__stdcall 或者 PASCAL 类型的。这种不确定性之所以会出现，其原因在于如果不知道函数的原型，就不能确定在堆栈中存放参数的顺序。但是，如果编译器是已知的，并且程序员使用的是默认的调用类型，那么就可以确定函数调用的类型。对于在 Windows 下运行的程序来说，PASCAL 与__stdcall 调用都用的很广泛，因此不确定性仍然存在。然而，如果可以同时得到调用函数与被调用函数的话，那么总是可以在被传递的参数与被接受的参数之间建立一种对应关系。目前本文没有考虑 PASCAL 类型的调用规范。

图 3-2 是一个 IDA 识别函数参数的例子。从源代码中可以很容易的分析出，函数 fun 有两个形式参数，一个是字符指针，一个是整型。从函数 main 中对 fun 的调用点处也可以确定实参的信息：参数 1 是字符数组变量 buff1，大小为 200 字节，参数 2 是立即数 10。因此，如果在函数 fun 中有对不安全库函数的调用，则可以通过 main 对 fun 的调用关系，将在 main 函数中调用 fun 时的字符数组的信息传递到 fun 函数中，从而可以检测由于此次调用是否可能导致栈缓冲溢出。对比反汇编后的代码，调用点处的实参信息就不如源代码中那么容易确定。调用点之前的 push 操作是否是将被调用函数的参数压栈，不能通过对代码的扫描确定。但是分析函数 sub_401000 可以发现，该函数有两个形参：arg_0 和 arg_4，因此，可以通过调用点处的 push 操作和函数形参的对应关系，确定调用 sub_401000 函数时传入的实参是 0Ah 和 var_130。正如前面所提到的，参数的传递方式不只通过 push 指令一种方式，因此，只通过 push 操作和函数形参的对应关系对分析结果可能会

有影响，这也是本文实现的原型系统需要改进的地方。

```
void fun(char *p, int i)
{
    char buff[50];
    int j;
    ......
}
int main()
{
    char buff1[200];
    char buff2[100];
    strcpy(buff2, buff1);
    fun(buff1, 10);
    return 0;
}
```

```
.text:00401000 sub_401000   proc near
.text:00401000 var_38          = dword ptr -38h
.text:00401000 var_34          = dword ptr -34h
.text:00401000 arg_0           = dword ptr  8
.text:00401000 arg_4           = dword ptr  0Ch
 ......
.text:0040101F sub_401000     endp
.text:00401020 _main        proc near
.text:00401020 var_130         = dword ptr -130h
.text:00401020 var_68          = dword ptr -68h
.text:00401020       push    ebp
.text:00401021       mov     ebp, esp
.text:00401023       sub     esp, 130h
.text:00401029       lea     eax, [ebp+var_130]
.text:0040102F       push    eax
.text:00401030       lea     ecx, [ebp+var_68]
.text:00401036       push    ecx
.text:00401037       call    _strcpy
.text:0040103C       add     esp, 8
.text:0040103F       push    0Ah
.text:00401041       lea     edx, [ebp+var_130]
.text:00401047       push    edx
.text:00401048       call    sub_401000
.text:0040104D       add     esp, 8
.text:00401050       xor     eax, eax
.text:00401052       mov     esp, ebp
.text:00401054       pop     ebp
.text:00401055       retn
.text:00401055 _main        endp
```

图3-2 IDA 识别参数的例子

3.1.3 局部堆栈变量

局部变量在堆栈之中进行分配，在函数执行完毕后删除。首先，任何传递给函数的参数都存放在堆栈之中。调用该函数的 call 指令将返回地址存放在参数的上方。一旦获取 CPU 控制权，函数就打开堆栈页面（也就是保存 EBP 寄存器以前的值，接着设置其值使它与指向栈顶的 ESP 寄存器值相等）。空闲堆栈区域位于

EBP 指向的地址的上方（也就是位于低端地址），而服务性数据（存储的 EBP 寄存器值以及返回地址）同参数一样位于它的下方。

位于栈顶指针（ESP 寄存器值）上方的堆栈区域可以被删除或者被丢弃。例如，硬件中断处理程序可以在程序中某个不能预见的地方或者时间段使用该堆栈区域。如果函数使用堆栈（保存寄存器的值或者传递参数），那么就会破坏堆栈。避免出现这种情况的方法是向上移动堆栈指针，直到占据了堆栈的这个区域为止。位于新的 ESP 地址“之下”的内存的完整性得以保证（从而避免了无意中的破坏）。对 push 指令的另一次调用将在不删除局部变量的情况下把数据存放在堆栈的顶部。

在函数执行结束时，将强行要求函数把 ESP 寄存器值返回到前一个位置。如果不能返回这个值，ret 指令将无法读取堆栈的返回地址；相反，它将读取“最上端”的局部变量值而控制传递到无法预知的地方。

图 3-3 的左半部分给出了函数被调用时刻的堆栈状况。函数打开堆栈页面，保存 EBP 寄存器的过去值，并将它设置成与 ESP 寄存器值相等。图 3-3 的右半部分显示了为局部变量分配 0x14 字节的堆栈内存情况。这是通过将 ESP 寄存器值向上移动而进入低地址区域来实现的。局部变量在堆栈中的分配就好像是通过 push 指令压入到这个位置一样。在函数执行完毕以后，函数增加 ESP 寄存器值并将值返回到它的前一个位置，以此释放局部变量所占据的内存。然后，函数从堆栈中恢复 EBP 寄存器的值，同时关闭堆栈页面。

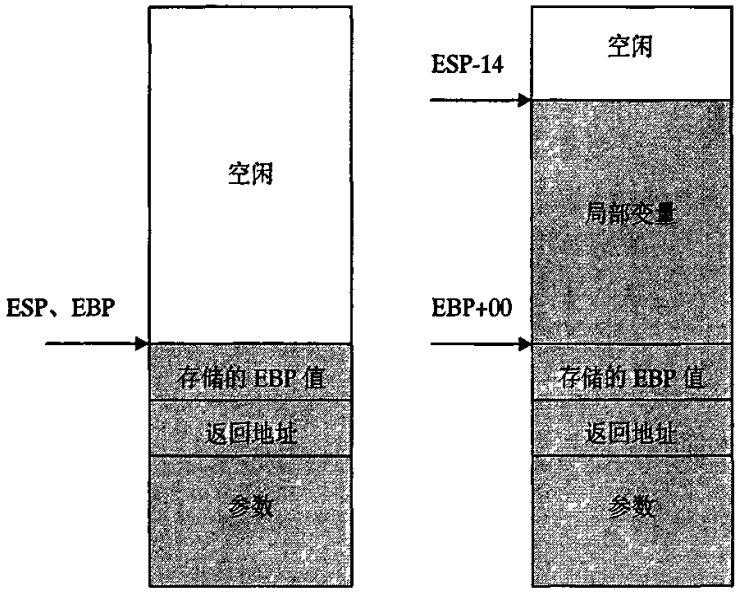

图3-3 在堆栈分配局部变量的机制

3.1.3.1 对局部变量进行寻址

局部变量与堆栈参数以相似的方式进行寻址。唯一不同的地方是，参数位于EBP的“下方”，而局部变量驻留在它的“上方”。换句话说，参数相对于EBP地址具有正的偏移量，而局部变量的偏移量是负的。例如，[EBP+xxx]表示的是一个参数，而[EBP-xxx]表示的则是一个局部变量。

指向堆栈页面的寄存器起到一个栅栏的作用：函数参数位于它的一侧，而局部变量位于另一侧，如图 3-4 所示。优化编译器能够通过 ESP 寄存器直接对局部变量与参数进行寻址，从而将 EBP 寄存器腾出来进行更有用的工作。

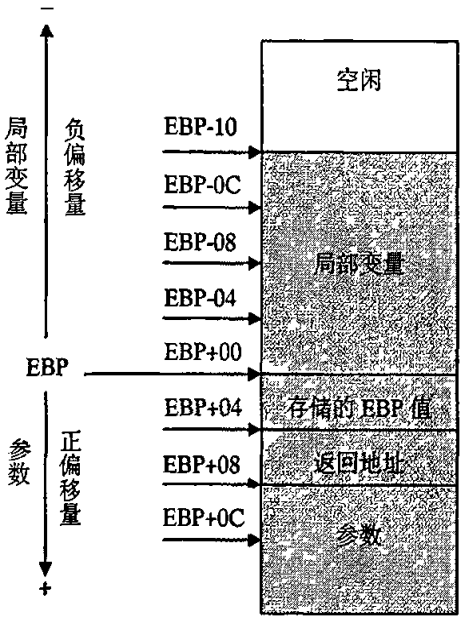

图3-4 对局部变量进行寻址

3.1.3.2 内存分配与删除

有许多方法可以用来为局部变量分配与删除内存。例如，sub esp, xxx 指令可以用于函数入口处，而 add esp, xxx 指令则可以用于函数出口处。在默认情况下，绝大多数的反汇编器将该数解释为一个很大的正整数。在分配一个很小的内存块时，优化编译器用 push reg 指令来取代 sub reg 指令，前者的指令代码要少几个字节。不过，这会给识别带来新的问题：这样的指令是在堆栈上保存寄存器的值、传递参数还是在为局部变量分配内存。

清除内存的算法也是存在二义性的。除了会因为 add esp, xxx 指令而使栈顶指针寄存器的值增加以外，还可以看到指令 mov esp, ebp。(在堆栈页面被打开的时候，虽然将 ESP 寄存器的值复制给 EBP 寄存器，但是在函数执行区间，EBP 寄存器的值不会发生改变)。最后，内存可能通过 pop 指令来释放，这条指令将局部变量一个接一个地从堆栈中弹出来而存放到任何没有使用的寄存器之中。

通过使用 sub 指令与 add 指令，内存的分配是一致的，并且关于内存的解释也清楚明了。如果内存是通过使用 push 指令来分配，并且通过 pop 指令来清除，那么这种内存创建行为就会与在堆栈上进行的简单寄存器分配与释放行为变得不可区分。作为一种复杂情况，函数还包含有用于分配寄存器的指令，并夹杂一些内

存分配指令。

分配的内存量可以通过函数体里面“最高的”局部变量所对应的偏移量来确定。换句话说，在所有的[EBP-xxx]表达式中，最大的 xxx 偏移量一般等于为局部变量分配的内存的字节数目。然而，局部变量可以声明但不使用。虽然为这些局部变量分配内存（尽管优化编译器将它们作为多余的变量而加以删除），但是却不存在对它们的引用。在这种情况下，用于计算要分配的内存量的算法所得到的值会显得太低。不过，这种错误并不对程序分析结果产生什么影响。

初始化局部变量的方法有两种：通过 mov 指令（诸如 mov [ebp-04], 0x666 指令）为变量赋予必要的值，或者使用 push 指令（诸如 push 0x777 指令）直接将值压入堆栈。这可以使局部变量的内存分配与它们的初始化过程顺利地结合在一起。

在大多数情况下，流行的编译器使用 mov 指令执行初始化操作，而某些特别的汇编器可能使用 push 指令，这样做的目的往往在于误导黑客而起到保护的作用。

3.1.3.3 结构体与数组

结构体是程序员用得特别多的数据结构形式。它可以用来将一些相关的数据组织在一起，从而使程序具有更好的可读性和可理解性。但是，结构体只存在于程序的源代码中，在程序的编译阶段，结构体几乎完全“溶解”在程序中，从而使得与那些从任何角度讲都不相关的普通变量无法相互区别。如图 3-5 所示的示例代码。从反汇编代码可以看到，结构体变量 y 被分解为普通的三个变量 var_18，var_8 和 var_4。而且在对函数 fun 调用时，传递的结构体指针在反汇编代码对应的是传递变量 var_18 的地址。结合被调用函数 fun 的反汇编代码，在过程 sub_401000 中对_printf 的调用时，有通过形参 arg_0 的寄存器寻址指令 mov edx, [ebp+arg_0] /mov eax, [edx+10h] /push eax 等。因此，能否根据调用函数和被调用函数的分析，确定调用函数中的 lea ecx, [ebp+var_18] /push ecx 指令传递的是结构体指针而不是普通变量的指针，值得进一步的研究。

```
struct zzz
{
    char s[16];
    int a;
    int b;
};
void fun(struct zzz *p)
{
    printf(……);
}
int main()
{
    struct zzz y;
    strcpy(y.s,"hello");
    y.a = 1;
    y.b = 2;
    fun(&y);
    return 0;
}
```

```
.text:00401000 sub_401000   proc near
.text:00401000 arg_0          = dword ptr  8
.text:00401003       mov    eax, [ebp+arg_0]
.text:00401006       mov    ecx, [eax+14h]
.text:00401009       push   ecx
.text:0040100A       mov    edx, [ebp+arg_0]
.text:0040100D       mov    eax, [edx+10h]
.text:00401010       push   eax
.text:00401011       mov    ecx, [ebp+arg_0]
.text:00401014       push   ecx
.text:00401015       push   offset unk_4060FC
.text:0040101A       call   _printf
.text:00401023 sub_401000    endp
.text:00401030 _main       proc near
.text:00401030 var_18         = dword ptr -18h
.text:00401030 var_8          = dword ptr -8
.text:00401030 var_4          = dword ptr -4
.text:00401030       push   ebp
.text:00401031       mov    ebp, esp
.text:00401033       sub    esp, 18h
.text:00401036       push   offset aHello
.text:0040103B       lea    eax, [ebp+var_18]
.text:0040103E       push   eax
.text:0040103F       call   _strcpy
.text:00401044       add    esp, 8
.text:00401047       mov    [ebp+var_8], 1
.text:0040104E       mov    [ebp+var_4], 2
.text:00401055       lea    ecx, [ebp+var_18]
.text:00401058       push   ecx
.text:00401059       call   sub_401000
.text:0040105E       add    esp, 4
.text:00401061       xor    eax, eax
.text:00401063       mov    esp, ebp
.text:00401065       pop    ebp
.text:00401066       retn
.text:00401066 _main          endp
```

图3-5 结构体的例子

结构体与数组以相邻的内存地址而连续的存放在堆栈之中。虽然较小的数组

索引位于较小的地址空间，但是可以通过相对于堆栈页面指针寄存器的较大偏移量来寻址，因为局部变量是通过一个[EBP-0x4]>[EBP-0x10]的负偏移量来寻址的。

因为 IDA 在为局部变量给出名称时会省去负号，因此混乱随之增多。例如，对于变量 var_4 与 var_10 来说，后者占据较小的地址，而它的索引值则较大。如果变量 var_4 与 var_10 位于数组的两端，直观的感觉将是变量 var_4 位于数组的开头，而变量 var_10 位于数组的末尾，尽管它们所处的位置刚好相反。

在某些情况下，一个结构体、数组乃至特殊变量都必须与取值为特定的 2 次幂的地址保持对齐。不过，栈顶指针值不是预先定义的，所以，计算机只是简单地将 ESP 寄存器值的低位加以丢弃。偶数值的较低位为零，为确保栈顶指针的值能够被 2 除尽而没有余数，只需简单地强制低位为零就可以了；如果有两个低位被设置为零，计算的结果将是 4 的倍数；如果有三个低位被设置为零，那么计算结果将是 8 的倍数。依此类推。在绝大多数情况下，使用 and 指令进行复位。例如，指令 and esp, FFFFFFF0 将使 ESP 寄存器的值成为 16 的倍数。

3.1.4 判断栈溢出的标准

我们定义栈缓冲区的整数范围为[Min，Max]，其中，Min 是指栈缓冲变量相对相邻且在栈下方的变量（即相对返回地址偏移量小的变量）的偏移，Max 指栈缓冲变量相对返回地址的偏移。如果存在两个栈缓冲变量：源缓冲区 src 和目的缓冲区 des，我们的判断标准可以确定为：

1. $Min_{des}>Max_{src}$，目的缓冲区足够容纳源缓冲区，不会溢出；
2. $Max_{des}<Min_{src}$，源缓冲区将覆盖函数的返回地址，导致溢出；
3. $Min_{des} < Min_{src} < Max_{des}$，源缓冲区与目的缓冲区范围有交叉，有可能导致与目的缓冲区相邻的变量值被修改，可能是不安全的。

以图 3-1 的程序为例，可以画出函数_main 的栈帧结构如图 3-6 所示：

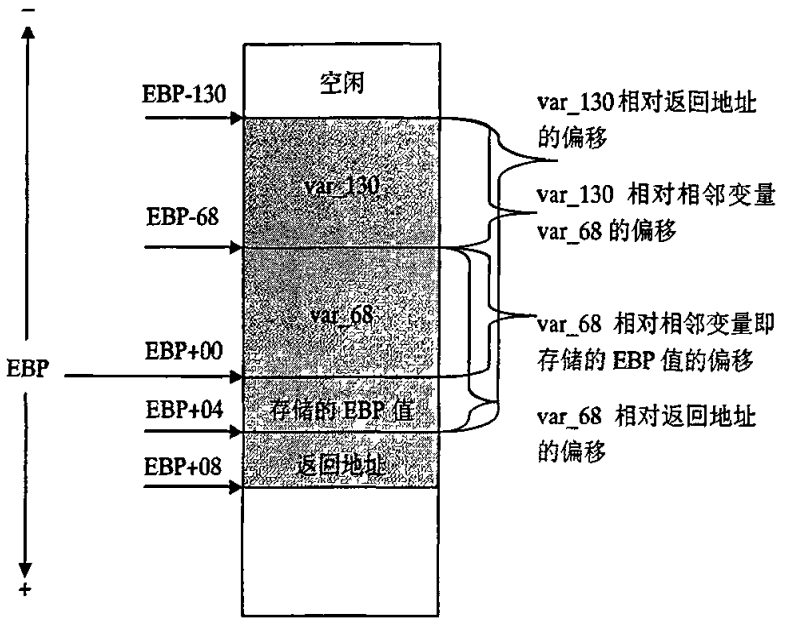

图3-6 _main 的栈帧结构

从图 3-6 可以计算出，栈缓冲变量 var_68 相对返回地址的偏移为 68h+4h=6Ch=108，相对相邻且在栈下方的变量即存储的 EBP 值的偏移为 68h=104，var_68 的整数范围为[104，108]。同理可以计算出栈缓冲变量 var_130 的整数范围为[130h-68h=C8h=200，130h+4h＝134h＝308]。因为 $Max_{var\_68}<Min_{var\_130}$，所以如果存在以 var_68 为目的缓冲区和以 var_130 为源缓冲区的字符串拷贝操作，则可以确定存在潜在的栈缓冲溢出危险。但是，由于是静态分析，并且没有进行控制流的分析，因此不能判断这个潜在的溢出能否被利用，这也是目前基于静态分析的自动发掘工具所要解决的一个大问题。

3.2 词法、语法分析基础

经过上述的分析可以发现，从二进制代码的反汇编代码中是可以找出与缓冲区溢出相关的语法特征的。如果利用相关的编译技术，对反汇编代码进行词法和语法分析，确定函数间的调用关系，那么就能够通过进一步的分析判断是否存在潜在的缓冲区溢出可能。

3.2.1 形式定义

本节给出一些后面将使用到的形式定义[48]。

定义 2.1 一产生式或重写规则（简称规则）是有序对(U，x)，通常写作

$$U ::= x 或 U \rightarrow x$$

其中，U 是符号，x 是有穷符号串。U 是产生式的左部，x 是其右部。

定义 2.2 文法 G[Z]是规则的非空有穷集合。Z 是一个符号，它至少需要在一条规则中作为左部出现，通常把这个符号称为“识别符号”（开始符号）。在所有的规则中，规则左部和右部中的所有符号组成一个集合，称为文法的字汇表 V。

定义 2.3 给定文法 G，把作为规则左部出现的那些符号称为非终结符号或语法成分，它们形成了非终结符号集合 V_n。而规则中不属于 V_n 的那些符号称为终结符号，它们形成了集合 V_t。

定义 2.4 令 G 是一文法，x 和 y 是定义在 V 上的符号串（可以是空串），如果 V 上存在另外两个符号串 v 和 w，且 v=xUy，w=xuy，若文法 G 中有规则 U::=u，那么有

$$v \Rightarrow w$$

称 v 直接推导出 w，或 v 直接产生 w，或 w 直接规约到 v。

定义 2.5 如果存在一直接推导序列

$$v = u_0 \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots\cdots \Rightarrow u_n = w$$

其中 n>0，那么，我们说 v 产生 w，或 w 规约到 v，并记做 $v \overset{+}{\Rightarrow} w$。

3.2.2 正则文法和正则表达式

3.2.2.1 文法和语言分类

乔姆斯基把文法定义为四元组，即：

$$G = (V, V_T, P, Z)$$

其中，V：符号集合；V_T：终结符号集合，$V_T \subset V$；P：有穷规则的集合；Z：识别符号，$Z \in V - V_T$。某一文法的语言是指能由 Z 出发，利用规则 P 推导出来的那些终结符号串所组成的集合，即：

$$L(G) = \{x \mid x \in V_T^+, Z \overset{+}{\Rightarrow} x\}$$

乔姆斯基根据文法规则中重写规则 P 的不同，把文法分成四种类型[48]。四种类型的文法对应着四种类型的语言。

1) 0 型文法

若 P 中具有如下形式的规则：

$$u ::= v$$

其中，$u \in V^+$，$v \in V^*$，则 G 为 0 型文法，或称“短语结构”文法。由 0 型文法所确定的语言为 0 型语言，可以由一种所谓图灵机（Turing）来接受。

2) 1 型文法

若文法 G 的 P 中有如下形式的规则：

$$xUy ::= xuy$$

其中，$U \in V_n, V_n = V - V_T$；$x, y \in V^*$；$u \in V^+$，则 G 为 1 型文法，也称为上下文敏感文法，即只有在上下文 x……y 中，才允许把 U 重写成 u。1 型文法所确定的语言为 1 型语言，可以由一种所谓线性界限的图灵机所接受。

3) 2 型文法

若 P 中规则具有如下形式：

$$U ::= u$$

其中，$U \in V_n$；$u \in V^*$，那么，这类文法为 2 型文法，也称为上下文无关文法，即把 U 重写为 u 时，无须考虑上下文。

4) 3 型文法

若 P 中具有如下形式的规则：

$$U ::= N \text{ 或 } U ::= WN$$

其中，$U \in V_n$；$N \in V_T$；$W \in V_n$，那么文法 G 称为 3 型文法或正则文法。由 3 型文法所确定的语言称为 3 型语言、或正则语言、或正则集合。上述定义的 3 型文法称作“左线性的”。因为若规则右部由一个非终结符和终结符组成的话，则非终结符在左边。类似的可以定义右线性的 3 型文法，其规则具有如下形式：

$$U ::= NW$$

由 3 型文法所确定的语言，可以用一种有穷状态自动机来接受。

3.2.2.2 正则表达式

表示正则语言的一种更简洁的方式是使用正则表达式。形式语言和自动机理论已经证明，正则文法和正则表达式是等价的。

正则表达式使用三种操作符：连接、选择以及重复。假定有两个正则表达式 e_1 和 e_2，它们分别产生语言 L_1 和 L_2，于是定义正则表达式的连接操作为 $e_1e_2 = \{xy \mid x \in L_1, y \in L_2\}$。选择操作可用“|”或“+”表示，且定义为 $e_1 \mid e_2 = \{x \mid x \in L_1 \text{或} x \in L_2\}$。重复运算用“{}”表示，表示大括号中的表达式的 0

次到若干次的自重复连接，即$\{e_1\}=\{x \mid x \in L_1^*, L_1^* = \bigcup_{i=0}^{\infty} L_1^i\}$。

所有正则表达式都可由下列规则构造出来[48]。

1. Φ是一个表示空集的正则表达式；
2. ε是一个正则表达式，它所表示的语言仅包含一个空符号串，即$\{\varepsilon\}$；
3. a是一个正则表达式，$a \in V_t$，它所表示的语言是由单个符号a所组成，即$\{a\}$；
4. 如果e_1和e_2是正则表达式，其表示的语言分别为L_1和L_2，则：
   1) $(e_1)|(e_2)$是一个表示语言$L_1 \cup L_2$的正则表达式；
   2) $(e_1)(e_2)$是一个表示语言L_1L_2的正则表达式；
   3) $\{e_1\}$是一个表示语言L_1^*的正则表达式。

如果我们用正则表达式来描述单词符号，那么就可以构造一个程序，它能够根据给定的正则表达式自动生成相应的词法分析程序。本文将在3.2.5节介绍这样的词法分析程序的自动生成器——LEX/FLEX。

3.2.3 相关语法分析方法

语法分析的任务是：按照文法，从源程序符号串中识别出各类语法成分，同时进行语法检查，为语义分析和代码生成做准备。按照构造分析树或语法树方式，算法可分成两大类，即自顶向下分析和自底向上分析。本章将在3.2.6节介绍的工具YACC是一个LALR(1)分析器的自动生成器，因此本节只对相关的自底向上语法分析方法加以介绍。

自底向上的分析方法，也称为“移进-规约”法，这种方法的一般过程是：设置一个寄存符号的先进后出栈，称为符号栈，用来记录分析的历史和指示分析的下一步动作。在分析进行时，把输入符号一个个的按扫描顺序移进栈里，当栈顶符号串形成一个句柄（为某条规则的右部）时，就进行一个规约，把栈顶构成句柄的那个符号串用相应规则左部的非终结符号来代替。接着再检查在栈顶是否又出现了新的句柄，若出现新的句柄，就再进行规约；若没有形成新的句柄，则再从符号输入串移进新的符号，……，如此继续到整个符号串处理完。最终如栈底为识别符号，则所分析的输入符号串为合法的符号串，报告成功；否则，是不合法的符号串，报告错误。

例如，对文法G[S]：

$S ::= aAcBe$

$A ::= b$

$A ::= Ab$

$B ::= d$

输入符号串为 abbcde，检查是否是该文法的合法句子。采用自底向上分析，即能否通过一步步规约当前句型的句柄，最终规约为识别符号 S。按照上面的方法，先设立一个符号栈。为实现上的方便，以“#”作为待分析的符号串的左右分界符。作为初始状态，先将符号串的左分界符推进符号栈，作为栈底符号。对符号串 abbcde 的分析过程如表 3-1 所示。

表3-1 符号串 abbcde 的分析

步骤	符号栈	输入符号串	动作
1	#	abbcde#	左分界符进栈
2	#a	bbcde#	进栈
3	#ab	bcde#	进栈
4	#aA	bcde#	用 $A ::= b$ 规约
5	#aAb	cde#	进栈
6	#aA	cde#	用 $A ::= Ab$ 规约
7	#aAc	de#	进栈
8	#aAcd	e#	进栈
9	#aAcB	e#	用 $B ::= d$ 规约
10	#aAcBe	#	进栈
11	#S	#	用 $S ::= aAcBe$ 规约
12	#S	#	接受

存在着多种自底向上的分析方法，但是，它们都是按同一种方式工作的，即都是按一般“移进-规约”法的基本原理建立起来的。其中较为常用的方法有算符优先分析法和 LR 分析法。LR(K)分析器根据符号栈的内容以及向前看输入串 K 个字符来决定分析动作，向前看的符号越多，则分析器越复杂。LR(0)分析器在确定分析动作时，不需要向前查看任何字符。虽然这种分析器不实用，但由于其简单性，可以作为构造更一般的 LR 分析器的起始点和基础。LR(1)、SLR(1)、和 LALR(1)都是向前查看一个符号的分析器。

3.2.4 属性翻译文法

3.2.6 节将介绍的 YACC 工具的文法规则和语义处理中，函数的形参、局部堆

栈变量、寄存器值的跟踪以及函数调用的实参等成分，利用了属性翻译文法的方法加以提取，因此属性翻译文法也是本文扫描工具实现的关键技术之一。

语法制导翻译文法是目前大多数编译程序普遍采用的一种方法。在这种方法中，将根据产生式所包含的语义，用一个或多个语义子程序（或称之为语义动作）所要完成的功能来描述，并将这些语义子程序插入到产生式的相应位置，从而形成称之为翻译文法的文法。当在语法分析过程中使用该产生式时，就可在合适的时间调用这些语义子程序完成所需的翻译。进一步，可分析文法中每个符号的语义，并将这些语义以属性的形式附加到相应的符号上，再根据产生式所包含的语义，给出符号间属性的求值规则，从而形成称之为属性翻译文法的文法。这样，当在语法分析中使用该产生式时，可根据属性求值规则对相应属性进行求值，从而完成翻译。YACC（Yet Another Compiler-Compiler）采用语法制导翻译的方法获取翻译和语义处理信息，成为语法分析程序的自动生成工具。

属性翻译文法是一个带有下列说明的翻译文法[48]：

1. 每个终结符、非终结符和动作符号都有一个与其相关的属性的有穷集，且每个属性都有一个值域。
2. 每一个非终结符号和动作符号的属性可分为两类：继承属性和综合属性。继承属性是一种按自顶向下、自左向右的求值规则求得的属性，综合属性则是一种通过自底向上进行求值的属性。
3. 继承属性的求值规则是：
   1) 开始符号的每个继承属性具有指定的初始值。
   2) 给定一产生式，该产生式右边符号的继承属性用该产生式其它符号的属性值进行计算。
4. 综合属性的求值规则为：
   1) 每一个输入符号的每一个综合属性具有指定的初始值。
   2) 给定一产生式，该产生式左部非终结符的综合属性用该产生式左部或右部某些符号的属性值进行计算。
   3) 给定一动作符号，其综合属性将用该动作符号的其它属性值进行计算。

说明的第 3 部分规定了继承属性的求值规则。这儿仅说明了产生式右部符号的计算继承属性值的规则。对产生式左部的非终结符，其继承属性则继承前面产生式中该符号已有的继承属性值。

说明的第 4 部分规定了产生式左部的非终结符以及动作符号的综合属性的求值规则。对终结符号其综合属性具有指定的初始值。在具体实现中，该初始值将

由词法分析程序提供。至于产生式右部的非终结符号的综合属性，则取当它是某个产生式的左部时所求得的综合属性值。

3.2.5 词法分析程序的自动生成器

3.2.5.1 LEX/FLEX 简介

LEX 是词法分析程序的生成器（Lexical Analyzer Generator），是在 1972 年由贝尔实验室在 UNIX 上首先设计实现，它是 UNIX 标准应用程序。1984 年的 GNU 工程推出 FLEX（Fast Lexical Analyzer Generator），它是对 LEX 的扩充，同时也与 LEX 兼容。目前，LEX/FLEX 已经可以在 UNIX、Linux、Windows 等环境运行，不仅高效率地为多种程序设计语言实现了词法分析器，而且在一些系统软件的开发过程中也得到了广泛的应用。由于 LEX/FLEX 是兼容的，后面将二者通称为 LEX。

以状态转换图为工具能够实现对单词的识别，而状态转换图是有限自动机的等价表示形式，并且为正则语言，其证明了正则式所表示的语言与有限自动机所识别的语言是完全等价的。一般程序语言单词的词法规则可以表示成正则文法，可以用正则式对其进行描述。由此，为词法分析器的自动生成奠定了理论基础。

词法分析程序生成器接收用正则式表示的定义在某语言字母表 Σ 上的单词，然后构造出与此正则式等价的非确定有限自动机 NFA M'，再对 M'进行确定化和化简，得到确定有限自动机 DFA M，则 M 就是所求的扫描器。LEX/FLEX 即是根据这一原理实现的。

LEX 系统包括 LEX 语言和 LEX 编译器两部分。LEX 语言用来描述单词的正则式，使用 LEX 语言描述的词法文件经 LEX 编译器编译后，输出词法分析程序。

3.2.5.2 LEX 源程序结构

一个 LEX 源程序由三部分组成：定义部分、规则部分和辅助函数部分，这三个部分由“%%”分隔，因此，其程序结构可以表示为：

定义部分
%%
规则部分
%%
辅助函数部分

其中，定义部分和辅助函数部分是任选的，规则部分是必须的。如果辅助函

数部分缺省，则第二个分隔号“%%”可以省去；但由于第一个分隔号“%%”用来指示规则部分的开始，故即使没有定义部分，也不能将其省去。

定义部分的作用是对规则部分要引用的文件和变量进行说明，通常可包含头文件表、常数定义、全局变量定义以及宏定义等。除宏定义外，它们与 C 语言程序的书写格式十分类似。每一个宏定义由分隔符（适当个数的空格或制表符）连接的宏名字和宏内容组成。例如语言中的字母可以定义为：Letter [a-zA-Z]，数字可以定义为：Number [0-9]。需要注意的是，凡对已定义的宏名字的引用，都需要用大括号将他们扩起来，以免发生混淆。除宏定义外，定义部分的其余代码需用符号“%{”和“%}”括起来。另外， LEX 源程序所使用的 C 语言库文件和外部变量，也应分别用#include 和 extern 予以说明，并置于上述括号之内。

在 LEX 源程序中，起标识作用的符号“%%”，“%{”以及“%}”都必须处于所在行的最左字符位置。另外，在其中也可随意添加 C 语言形式的注释。

规则部分由一组识别规则组成，其书写格式为：

模式 R1　　动作 A1

模式 R2　　动作 A2

……

模式 Rn　　动作 An

其中模式 Ri 是正则表达式，用来描述单词的词型；动作 Ai 是 C 代码，与匹配的模式对应，用来指明从输入字符串中识别出词型为 Ri 的单词时，扫描器应执行的操作。每个模式都必须从所在行的最左字符位置开始书写，并用分隔符（适当个数的空格和制表符）与其后的动作分开。每个动作 Ai 可引用已定义的符号常量、全局变量和外部变量，并能调用辅助函数部分所定义的函数，必要时也可在动作中定义自己的局部变量。

通常使用的 LEX 模式定义如表 3-2 所示。

表3-2 常用 LEX 模式定义

模式	说明	示例
x	匹配单个字符 x	A
[abc]	匹配 a 或 b 或 c	[a-zA-Z]
[^abc]	匹配除去 a，b，c 之外的任意字符	[^ab0-9]表示匹配除小写字母 ab、数字 0-9 以外的任意字符
\	用来转义元字符。同样用来覆盖字符在此表中定义的特殊意义，只取字符的本意	
.	匹配除去换行符之外的任意字符	
r*	r 是正则式，r*匹配 0 个或多个 r	
r+	r 同上，r +匹配 1 个或多个 r	
r?	r 同上，r?匹配 0 个或 1 个 r	
r{2，5}	r 同上，匹配 2 到 5 之间次数的 r	
r{2，}	r 同上，匹配 2 次或更多次 r	
r{2}	r 同上，匹配 2 次 r	
{name}	name 是在定义部分出现的宏名	
“hello”	匹配字符串“hello”	
r|s	匹配正则式 r 或 s	
rs	匹配正则式 r 和 s 的连接	
……	……	……

辅助函数部分包含了识别规则动作代码段中所调用的各个局部函数，这些函数由用户编写，它们将由 LEX 系统直接拷贝到输出文件 lex.yy.c 之中。

3.2.5.3 LEX 的应用过程

下面以 FLEX 为例，说明 LEX 的应用过程。

首先，按照 FLEX 语言的格式要求，编写 FLEX 源程序。编写完的 FLEX 源程序需由 FLEX 系统的翻译程序进行处理。作为处理的结果，将输出一个名为 lex.yy.c 的 C 语言程序文件。此文件含有两部分内容，一是根据正则式构造的状态转移表，二是用来进行词法分析的驱动程序 yylex()，同时，FLEX 源程序中所列的语义动作也被直接拷贝到 lex.yy.c 之中。最后，再用 C 编译程序对 lex.yy.c 进行编译，并将支持扫描器运行的有关库函数（它们都包含在文件 flexlib.lib）连入目标

码程序，便得到了所要生成的词法分析程序。而后，在编译程序运行时，每从主程序调用 yylex()一次，就从被编译的输入字符流中识别一个单词。显然，对不同的 FLEX 源程序而言，FLEX 系统所生成的扫描器的驱动程序 yylex()都是相同的，仅状态转移表不同。

FLEX 工作过程如图 3-7 所示。

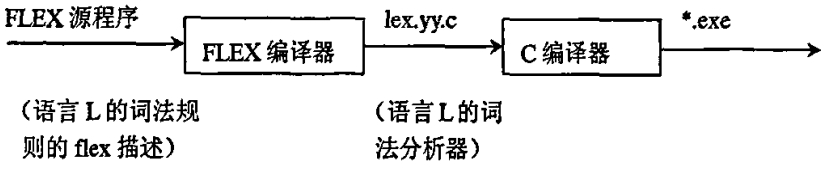

图3-7 FLEX 的工作过程

可以通过两种方式来使用 FLEX，一种是将 FLEX 作为一个单独的工具，用以生成所需的识别程序，而这些识别程序通常都出现在一些非开发编译器的应用领域中，诸如编辑器设计、命令行解释、模式识别、信息检索以及开关系统等。第二种方式是将 FLEX 和语法分析器自动生成工具（例如 YACC）结合起来使用，以生成一个编译程序的扫描器和语法分析器。

3.2.6 语法分析程序的自动生成器

3.2.6.1 YACC 简介

YACC（Yet Another Compiler-Compiler）是一个 LALR(1)分析器的自动生成器。YACC 与 LEX 一样，是贝尔实验室在 UNIX 上首先实现的，而且与 LEX 有直接的接口，它是 UNIX 的标准应用程序。GNU 工程推出 Bison，是对 YACC 的扩充，同时也与 YACC 兼容。目前 YACC 与 LEX 一样，可以在 UNIX、Linux、Windows 等环境运行。

YACC 是一个通用的语法分析生成器，它读入一个 LALR(1)上下文无关文法，生成一个用来分析这个文法的语法分析器，通常生成的语法分析器是一个 C 程序。只要熟练掌握 YACC 这个强有力的工具，就可以很容易开发出大多数语法分析器，如简单一点的计算器和复杂一点的程序设计语言。

自底向上语法分析的关键是构造出识别文法活前缀的状态机。LALR(1)文法的状态机的构造有两种途径，一种是先构造文法的 LR(1)状态机，然后通过同心状态合并的方法得到 LALR(1)状态机，YACC 采用的就是这种方法；另一种是先构造

LR(0)的状态机，再通过展望符传播的方式得到 LALR(1)状态机，Bison 采用的是这种方法。LALR(1)状态机的自动构造是语法分析生成器自动构造的理论基础。

3.2.6.2 YACC 源程序结构

YACC 语言和源程序是对语言的语法规则的描述，用来输入文法规则。YACC 源程序结构与 LEX 类似，也由三部分组成：声明部分、语法规则部分、辅助 C 代码部分，各部分之间用符号“%%”隔开，其结构如下：

```
%{
C 语言声明部分
%}
YACC 声明部分
%%
语法规则部分
%%
辅助 C 语言代码部分
```

其中声明部分又包含两部分内容：C 语言声明部分，该部分用符号“%{”和“%}”括起来；文法符号（一般为终结符）和文法规则的说明以及对文法规则说明的一些限定规则和条件的声明部分（如运算符的结合性和优先级等），该部分的每一项均以“%”开头。

语法规则部分是 YACC 源程序的主体部分，是对文法的全部规则及每一规则相关的语义动作的描述。如文法中某一文法规则：

<左部文法符号>：<候选式 1>|<候选式 2>| … |<候选式 n>

用 YACC 描述的一般形式为：

```
<左部文法符号>：<候选式 1>   {语义动作 1}
                |<候选式 2>   {语义动作 2}
                | …
                |<候选式 n>   {语义动作 n}
```

其中左部文法符号即非终结符要用小写字符串来表示，文法规则描述的候选式中，对文法形如+，-，*，/的单字符终结符要用单引号括起来，其它多字符终结符要用%token 来声明，并且用大写字符串表示。如‘+’，‘-’，INTEGER，IDENTIFIER，IF 和 RETURN 等都是终结符；expr，stmt 和 declaration 等都是非终结符。

语义动作是完成语义处理的 C 语言程序。一旦有规则匹配，这个规则所包含的语义动作就得到执行。通常，语义动作用来计算整条规则的语义值。例如，表达式规则可以写成如下形式：

```
expr: expr '+' expr {$$ = $1 + $3; };
```

规则后面用大括号括起来的部分定义的就是这条规则的语义动作。$j 表示规则右边（':' 右边）第 j 个终结符或非终结符的语义值，$$表示整个规则的语义值。这个规则的语义动作定义了加法表达式的值为加号左右两个表达式的值之和。

YACC 程序的第三部分，即辅助子程序部分，是由若干 C 语言函数构成的，如词法分析程序 yylex 及错误诊断程序等。

3.2.6.3 YACC 的应用过程

使用 YACC 自动构造语法分析器的过程如图 3-8 所示，YACC 编译器接收 YACC 源程序*.y，产生一个*.tab.c 的 C 程序，该程序就是生成的语法分析器，经过 C 编译器编译之后，再连接目标文件，生成可执行程序。

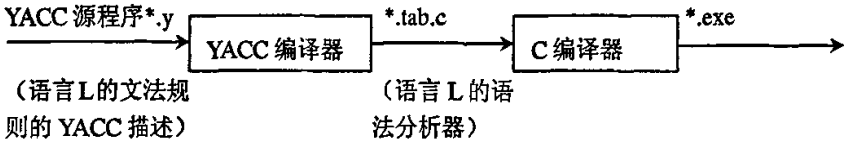

图3-8 YACC 的工作过程

在使用 YACC 之前，需要建立 YACC 语法文件，建立过程分为如下几步：

1. 用 YACC 语法书写 LALR(1)语法规则；
2. 编写词法分析器；
3. 编写 main()函数；
4. 编写错误处理程序。

建立了 YACC 语法文件以后，将其生成可执行代码，分为如下几步：

1. 用 YACC 程序将 YACC 语法文件生成语法分析程序；
2. 编译 YACC 语法分析程序，以及其它附属程序；
3. 连接目标文件，生成可执行程序。

3.3 小结

本章结合一些简单的 C 语言例子程序，利用反汇编工具 IDA 反汇编相应的二

进制文件，通过源代码和反汇编代码的对比，研究如何从二进制代码中识别高级语言的关键结构，并结合相关的词法、语法分析的理论基础，介绍了两个自动生成器：LEX 和 YACC。下一章将在本章的分析基础之上，实现一个针对二进制代码的栈缓冲溢出静态检测原型系统。

第四章 二进制代码的静态扫描设计与实现

通过前面几章的分析，我们可以发现：缓冲区的开辟和不安全的 C 库函数在二进制代码中都可以找到相应的语法特征，这些特征表现出来就是一系列的汇编指令。因此，通过分析二进制文件的反汇编代码是可以发现某种类型的缓冲区溢出漏洞的。本章将详细介绍基于本文第二章提出的栈缓冲溢出检测方法实现的二进制代码的静态扫描原型系统。

4.1 系统总体结构

系统主要分为两个模块：二进制代码解析模块和栈溢出漏洞静态分析模块，如图 4-1 所示。

首先使用 IDA 反汇编工具反汇编 Windows 下的 PE 格式二进制文件，生成.lst 的反汇编文件。解析模块以.lst 文件作为输入，调用词法、语法分析程序，对反汇编代码进行操作语义的分析，生成函数调用关系图。

在对反汇编代码进行操作语义的分析后，静态分析模块根据函数调用关系图进行栈缓冲溢出漏洞的分析，输出分析结果。

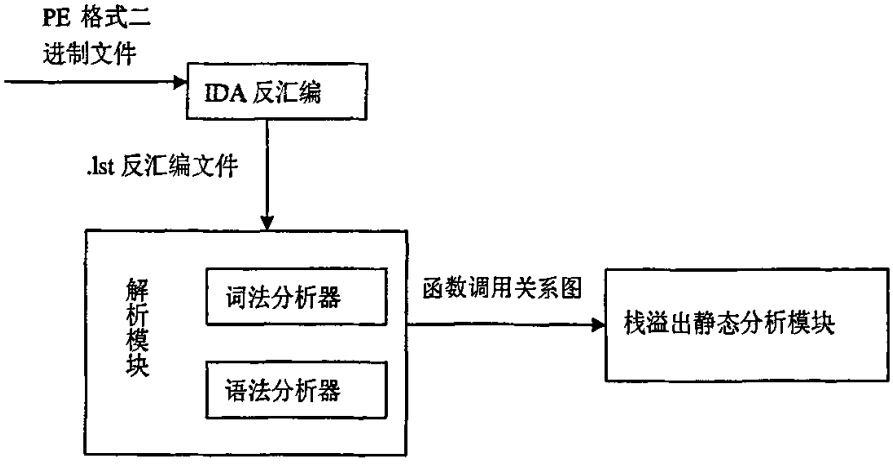

图4-1 系统总体结构

4.2 关键的数据结构

在整个代码解析模块中，核心工作是创建与函数对应的描述数据结构。该数据结构包含如下信息：

1. 实参、局部变量、形参结构定义

```
typedef struct tagParams
{
    string sName;              //变量名
    string sRefName;           //该变量引用的变量名
    string sType;              //变量类型：byte dword qword fword tbyte 等
    bool bRefed;               //判断是否曾经被另一个局部变量引用过，如果是，该
                               //局部变量很可能是被某个指针指向的起始地址
    vector<string> vRefedName; //保存曾经引用过的局部变量名
    int nType;                 //类型 1:局部变量 2:形参 3:整形常量 4:字符串常量
    int nOffset;               //局部变量在栈中相对返回地址的偏移
    int nMaxSize;              //栈缓冲的最大值
    int nMinSize;              //栈缓冲的最小值
}PARAMS, *PPARAMS, ARGUS, *PARGUS, LOCVARS, *PLOCVARS;
```

2. 函数调用结构定义

```
typedef struct tagCalledProc
{
    string sProcName;          //函数调用名
    int nLine;                 //函数调用在文件中的位置
    vector<ARGUS> vArgus;      //实参向量
}CALLEDPROC, *PCALLEDPROC;
```

3. 函数定义单元

```
class CProcUnit
{
private:
    string m_sProcName;            //函数名
    string m_sVirAddrStart;        //函数定义开始虚地址
    string m_sVirAddrEnd;          //函数定义结束虚地址
```

```
    int m_nVirAddrStartLine;            //开始虚地址在文件中的位置
    int m_nVirAddrEndLine;              //结束虚地址在文件中的位置
    vector<CALLEDPROC>      m_vRefNormalProc; //普通函数调用 vector
    vector<CALLEDPROC>      m_vRefDangerProc; //非安全函数调用 vector
    vector<string>          m_vReferedProc;   //调用本函数的函数名 vector
    vector<LOCVARS>         m_vLocVars;       //局部变量 vector
    vector<PARAMS>          m_vParams;        //形参 vector
    vector<string>          m_vInfo;          //漏洞输出信息
}
```

这样，通过一遍扫描，可以建立所有函数的描述表。扫描结束后，遍历该描述表，根据调用关系、局部变量、实参、写缓冲代码块，确定缓冲溢出发生的函数体。

4.3 重要的文法符号

下面列举出代码解析模块中语法分析器使用到的一些重要的文法符号。

1. 标识符和常量

   IDENTIFIER CONSTANT

2. 指令

   OP_ADD OP_SUB

   OP_PUSH OP_POP OP_INC OP_TEST OP_JNZ

   OP_LEA OP_MOV

   OP_CALL

3. 重复字符串操作指令

   OP_REP OP_REPNE

4. 字符串操作指令

   OP_LODS OP_LODSB OP_LODSD OP_LODSW

   OP_STOS OP_STOSB OP_STOSD OP_STOSW

   OP_MOVS OP_MOVSB OP_MOVSD OP_MOVSW

   OP_INS OP_INSB OP_INSD OP_INSW

   OP_OUTS OP_OUTSB OP_OUTSD OP_OUTSW

   OP_CMPS OP_CMPSB OP_CMPSD OP_CMPSW

OP_SCAS OP_SCASB OP_SCASD OP_SCASW

5. 操作数

OPND_VAR OPND_ARG

6. 关键字

K_OFFSET K_PTR K_PROC K_ENDP K_SHORT K_NEAR K_FAR K_LARGE

7. 寄存器

R_EAX R_EBX R_ECX R_EDX R_ESP R_EBP R_ESI R_EDI

R_AX R_BX R_CX R_DX R_SP R_BP R_SI R_DI R_AL R_AH R_BL R_BH

R_CL R_CH R_DL R_DH

R_ES R_CS R_SS R_DS R_FS R_GS

S_TEXT S_DATA S_RDATA S_IDATA

8. 数据类型

T_BYTE T_WORD T_DWORD T_QWORD T_FWORD T_TBYTE

T_DB T_DW T_DD T_DQ T_DF T_DQ T_DT

4.4 代码解析模块实现

4.4.1 词法分析及处理

词法分析器作为一个独立的子程序由语法分析程序调用。它的主要功能如下：

1. 跳过反汇编程序中的空白；
2. 从反汇编程序中识别单词符号，并向语法分析程序返回一个与单词符号相应的文法符号，如普通标识符 IDENTIFIER 或寄存器 R_EAX；
3. 跳过不处理的汇编指令。例如位操作指令、条件设置指令等。

尽管语法分析器只关心单词符号的类型，而且词法分析器也只需将单词符号的类型返回给语法分析器，但是符号的语义值对程序仍然非常重要，所以在词法分析阶段，要由词法分析器 yylex()生成每个终结符的语义值（非终结符的语义值是在语法分析阶段的语义动作中完成的）。全局变量 yylval 用于设定当前终结符的语义值，它的类型由 YYSTYPE 宏指定，在本文中，我们将它定义为标准 C++的 string 类型。

例如本文实现的扫描工具，当在词法分析器中满足正则表达式[0-9a-fA-F]+[hH]时，词法分析器返回一个十六进制常量类型 CONSTANT 给语法分析器，并将该十六进制常量的字符串表示作为 CONSTANT 的语义值，赋值给全局变量 yylval。语

法分析器识别出 CONSTANT 符号时，就可以访问 CONSTANT 的语义值。

4.4.2 语法分析及处理

语法分析程序对反汇编程序进行分析，为每一个识别出来的函数模块生成一个 CProcUnit 对象，分析每个函数模块的参数、局部变量以及在该函数模块中的函数调用，并确定这些函数模块之间的调用关系。

例如对于图 4-2 的反汇编代码及栈帧结构：

```
.text:00401000 sub_401000  proc near
.text:00401000 var_2D0  = byte ptr -2D0h
.text:00401000 var_26C  = dword ptr -26Ch
.text:00401000 var_DC   = dword ptr -0DCh
.text:00401000 var_74   = dword ptr -74h
.text:00401000 var_70   = byte ptr -70h
.text:00401000 var_8    = dword ptr -8
.text:00401000 var_4    = dword ptr -4
.text:00401000 arg_0    = dword ptr  8
.text:00401000      push  ebp
.text:00401001      mov   ebp, esp
.text:00401003      sub   esp, 2D0h
.text:00401009      mov   eax, dword_407030
.text:0040100E      mov   [ebp+var_8], eax
.text:004010B4 sub_401000       endp
```

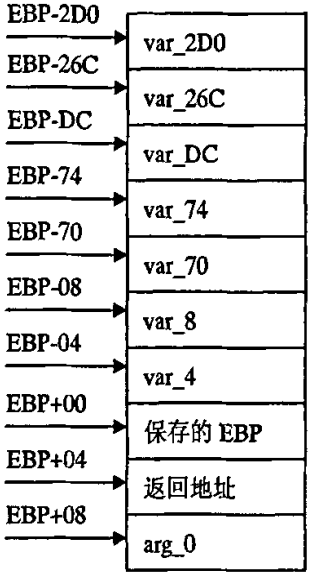

图4-2 一个例子程序

语法分析的任务就是：

1. 识别函数，如 sub_401000，并分配一个 CProcUnit 函数描述对象；
2. 识别局部堆栈变量及其整数范围，如 var_4、var_8 等，保存到 CProcUnit 对象的 m_vLocVars（局部堆栈变量向量）成员变量中；
3. 识别函数形参，如 arg_0 等，保存到 CProcUnit 对象的 m_vParams（形参向量）成员变量中；
4. 识别该函数中的普通函数调用及其实参，保存到 CProcUnit 对象的 m_vRefNormalProc（普通函数调用向量）成员变量中；
5. 识别 strcpy 之类不安全的函数调用及其实参，保存到 CProcUnit 对象的 m_vRefDangerProc（不安全函数调用向量）成员变量中；

6. 为了给出尽可能详尽的输出信息，可以相应保存函数名、文件中的位置等信息。

我们对语法规则的书写有这样的约定：终结符都由大写字母组成，非终结符都由小写字母组成。

4.4.2.1 函数

函数的文法规则可以描述为：

```
function
    : func_head func_body
    ;
func_body
    : decl_statements statements func_end
    | decl_statements func_end
    | statements func_end
    | func_end
    ;
func_head
    : K_PROC K_NEAR
    ;
func_end
    : IDENTIFIER K_ENDP
    ;
```

上述文法规则包含六个非终结符。其中函数体 func_body 包含四条规则，可以解释成：有局部变量和代码的函数、有局部变量，但没有代码的函数、只有代码的函数以及空函数。

函数头规则 func_head 表示已经识别到一个函数定义，所以只需要为 func_head 添加语义动作：生成一个 CProcUnit 对象；

为函数体规则 func_body 的第一条规则添加语义动作：将识别出来的局部变量信息以及 statements 中包含的函数调用信息添加到 CProcUnit 对象的成员变量中。对于其余三条规则，可以不做处理，因此只需要添加语义动作：删除 CProcUnit 对象即可。

4.4.2.2 函数的形参和局部堆栈变量

一个函数总的栈空间可以从 sub esp, 立即数和 add esp, 立即数指令中计算得到，而每个区域又是通过该函数的 ebp/esp 基址寄存器加减偏移量的引用得到。例如，如果函数内部有一条 lea eax, [ebp-274h]形式的指令，那么可以判断 ebp-274h 就是一个栈分配区域，这样也就确定了函数栈空间有一个 var_274 的变量。分析完函数中所有的这些类似的指令后，也就确定了函数的栈空间分配情况。由于 IDA 反汇编器已经做过这样的分析了，所以我们可以直接分析 IDA 生成的栈空间分配信息。如：

```
.text:00401000 sub_401000       proc near
.text:00401000 var_2D8          = byte ptr -2D8h
.text:00401000 var_274          = dword ptr -274h
.text:00401000 var_E4           = dword ptr -0E4h
.text:00401000 var_7C           = dword ptr -7Ch
.text:00401000 var_78           = byte ptr -78h
.text:00401000 var_C            = dword ptr -0Ch
.text:00401000 var_8            = dword ptr -8
.text:00401000 var_4            = dword ptr -4
.text:00401000 arg_0            = dword ptr 8
```

可以看到，现在栈中有 8 个区域，从 var_4 到 var_2D8。如果要确定 var_274 的大小，就是 274h-0E4h=190h 字节，而 var_274 相对栈基址的偏移就是 274h。

栈基址和 arg_0 之间有 8 个字节的偏移，通过分析函数中的第一条指令可以确定这 8 个字节中的内容。例如上面 sub_401000 函数的第一条指令如果是：

```
.text:00401000                  push    ebp
```

则可以判断相对栈基址偏移为 0h 的 4 个字节保存的是调用 sub_401000 函数的外层函数的栈基址（也就是 push 指令所保存的 ebp），而相对栈基址偏移为 4h 的 4 个字节就是函数的返回地址。所以这时 var_274 相对返回地址的偏移应该是 274h+4=278h。

形参和堆栈变量的文法规则可以描述为：

```
decl_statements
    : OPND_VAR '=' type K_PTR imm
    | OPND_ARG '=' type K_PTR imm
```

| IDENTIFIER '=' type K_PTR imm
| IDENTIFIER '=' IDENTIFIER K_PTR imm
| OPND_VAR '=' IDENTIFIER K_PTR imm
| OPND_ARG '=' IDENTIFIER K_PTR imm
| decl_statements OPND_VAR '=' type K_PTR imm
| decl_statements OPND_ARG '=' type K_PTR imm
| decl_statements IDENTIFIER '=' type K_PTR imm
| decl_statements IDENTIFIER '=' IDENTIFIER K_PTR imm
| decl_statements OPND_VAR '=' IDENTIFIER K_PTR imm
| decl_statements OPND_ARG '=' IDENTIFIER K_PTR imm
;

上述文法规则由六条普通规则和六条左递归规则组成，与形参相关的文法规则是第 2、6、8 和 12 条，其语义动作就是在函数描述结构 CProcUnit 对象中为形参数组添加一条形参记录。形参与实参的对应关系将在每个函数的调用点处得到更新。其余规则都是局部堆栈变量的规则，语义动作是在 CProcUnit 对象的堆栈变量数组中添加一条堆栈变量的记录，并计算该堆栈变量与返回地址之间的偏移和与在该堆栈变量下方的相邻变量之间的偏移。这两个偏移值就是该堆栈变量的范围属性。例如对于两个堆栈变量 var1 和 var2，分别具有范围属性[a，b]和[c，d]。var1 作为目的缓冲区，var2 作为源缓冲区，判断溢出的标准就是：

1. a>d，目的缓冲区足够容纳源缓冲区，不会溢出；
2. b<c，源缓冲区将覆盖函数的返回地址，导致溢出；
3. a<c<b，有可能导致溢出。

当函数中有指针和引用的定义时，利用上述办法确定栈空间的分配会出现误报的情况。例如对于下面的代码：

```
char buff[600];
char *p = buff + 100;
```

反汇编后的代码是：

```
.text:00401011                  lea     eax, [ebp+var_274]
.text:00401017                  mov     [ebp+var_7C], eax
......
.text:0040102D                  lea     ecx, [ebp+var_2D8]
```

根据上述办法，会得到 var_7C、var_2D8 和 var_274 三个缓冲区域。由于指针

p 的定义，导致 var_2D8 区域被一分为二。所以这时确定 var_2D8 的缓冲大小时就出现误报。

因此，如果结合 lea 和 mov 两条指令对缓冲区分配做分析，就可以有效减小这种误报对漏洞分析的影响。

4.4.2.3 函数调用的实参

在实现该原型系统时，只考虑了使用 push 指令传递函数实参一种方式。正如在 3.1.2 节所述，函数参数的传递方式不只使用 push 指令这一种方式，所以目前的实现方法存在一定的缺陷。

push 指令的操作数可以有三种方式：寄存器（reg），存储器（mem）和立即数（imm）。在实现时，我们将局部堆栈变量操作数（var_opnd）的文法规则和函数形参操作数（arg_opnd）的文法规则从存储器文法规则中单独分离出来处理。因此，push 指令的文法规则可以描述为：

```
push_statement
    : OP_PUSH reg32
    | OP_PUSH mem
    | OP_PUSH imm
    | OP_PUSH var_opnd
    | OP_PUSH arg_opnd
    ;
```

将局部堆栈变量和函数形参的文法规则从存储器的文法规则中分离出来，那么在添加 OP_PUSH var_opnd 和 OP_PUSH arg_opnd 两条文法规则的语义动作时，可以明确的标明：当前压栈的操作数是局部堆栈变量和函数形参。当在语法分析处理完，进行进一步的静态分析时，可以通过这个标记，判断是否需要通过函数调用图向上递归求取调用点处的确切实参信息：如果是局部堆栈变量，则不需要；反之，如果是形参，则需要根据被调用信息，求出该调用点的确切实参信息，才能获得缓冲区的整数范围。

当 push 指令的操作数是寄存器时，为了确定寄存器中存储的目标操作数，有必要对传送指令 mov 和取有效地址 lea 指令作操作语义的分析。特别是当在 C 程序中有局部堆栈指针和引用时，其相应的汇编代码也会涉及到 mov 和 lea 指令对局部堆栈变量操作数 var_opnd 和函数形参 arg_opnd 的操作。mov 指令和 lea 指令的文法规则可以描述为：

```
mov_statement
    : OP_MOV reg32 ',' reg32
    | OP_MOV reg32 ',' mem
    | OP_MOV reg32 ',' var_opnd
    | OP_MOV reg32 ',' arg_opnd
    | OP_MOV var_opnd ',' reg32
    ;
lea_statement
    : OP_LEA reg32 ',' mem
    | OP_LEA reg32 ',' var_opnd
    | OP_LEA reg32 ',' arg_opnd
    ;
```

通过对 mov 指令和 lea 指令进行操作语义的分析，可以确定寄存器 reg32 或局部堆栈变量操作数 var_opnd 中保存的实际操作数。因此，当进行 push 指令的操作语义分析时，就可以获得实际操作数的最终来源，并确定其整数范围。

4.4.2.4 优化代码的分析

VC 编译器可以对程序进行优化以提高执行速度，有些函数如 strcpy、strcat 等就被硬编码到目标文件中。本文分析了编译器对 strcpy 函数的优化。

1. VC7 优化代码的分析

VC7 编译器对 strcpy 函数的优化可以分为两种情况：当 strcpy 的两个参数均是局部变量的情况和 strcpy 的任何一个参数是调用函数的形参的情况。如下所示：

1) 两个参数均是局部变量

```
.text:00401070        mov    cl, byte ptr [esp+eax+134h+var_CC]
.text:00401074        mov    byte ptr [esp+eax+134h+var_130], cl
.text:00401078        inc    eax
.text:00401079        test   cl, cl
.text:0040107B        jnz    short loc_401070
```

2) 某个参数是形参

```
.text:00401035        mov    cl, [eax]
.text:00401037        mov    [edx+eax], cl
.text:0040103A        inc    eax
```

```
.text:0040103B          test    cl, cl
.text:0040103D          jnz     short loc_401035
```

因此，strcpy 优化后的汇编指令中涉及目的串和源串的指令可以用下面四条语法规则描述：

```
OP_MOV reg8     ',' mem
OP_MOV reg8     ',' var_opnd
OP_MOV mem      ',' reg8
OP_MOV var_opnd ',' reg8
```

其中 reg8 是 8 位寄存器的语法规则，var_opnd 是局部变量的语法规则，mem 是[eax]，[edx+eax]等形式的存储器寻址的语法规则。观察上面的优化指令可知，确定 strcpy 的目的串和源串的地址，关键是跟踪 32 位通用寄存器和局部堆栈变量的值。因此，通过对以 32 位通用寄存器和局部变量为操作数的汇编指令进行操作语义分析，可以确定目的串和源串的大小，从而判断 strcpy 函数调用是否可能导致栈缓冲溢出。如下面的语法规则所示：

```
OP_LEA reg32    ',' var_opnd
OP_LEA reg32    ',' arg_opnd
OP_MOV reg32    ',' arg_opnd
OP_MOV reg32    ',' var_opnd
OP_MOV reg32    ',' reg32
OP_MOV var_opnd ',' reg32
```

2. VC6 优化代码的分析

VC6 编译器对 strcpy 优化后的汇编代码如下所示：

```
.text:00401012          mov     edi, edx
.text:00401014          repne scasb
.text:00401016          not     ecx
.text:00401018          sub     edi, ecx
.text:0040101A          lea     ebx, [esp+2C8h+var_258]
.text:0040101E          mov     eax, ecx
.text:00401020          mov     esi, edi
.text:00401022          shr     ecx, 2
.text:00401025          mov     edi, ebx
.text:00401027          lea     ebx, [esp+2C8h+var_1F4]
```

```
.text:0040102E          rep movsd
.text:00401030          mov     ecx, eax
.text:00401032          xor     eax, eax
.text:00401034          and     ecx, 3
.text:00401037          rep movsb
```

观察上述汇编指令，可以判断 strcpy 优化代码的关键在于三条重复字符串操作指令：

```
repne scasb        //扫描源串长度
rep   movsd        //字符串拷贝
rep   movsb        //字符串拷贝
```

而这三条指令涉及的寄存器是固定的，即：字符串扫描指令 scasb 所扫描的源串地址存储在 edi 寄存器中；字符串传送指令 movsd、movsb 将 esi 所指向的双字、字节传送到 edi 所指向的内存单元。

所以，对这三条指令进行语义分析，并结合 VC7 代码优化分析相同的跟踪寄存器的方法，可以确定 strcpy 函数调用的目的缓冲区和源缓冲区，从而判断 strcpy 是否可能导致溢出。

4.4.3 建立函数调用图

模块化一直被看成是结构化编程的优点，因为将程序适当地分解成若干过程(函数)，有助于创建结构良好且易于理解和维护的程序。通过对代码的解析，已经生成了单独的函数描述结构，每个这样的结构都包含了该函数的局部变量列表，其中包括所有局部变量的变量名、相对栈基址的偏移、最大长度（由各偏移间接算出）、最小长度；函数的形参信息；该函数体内的函数调用信息，其中包括每个函数调用的描述（包括实参名，个数以及调用函数的始址）等等。由于对代码的一次扫描只获得了函数间的调用关系，所以下一步就是根据这些函数的调用关系建立其被调用关系，从而构造完整的函数调用图。

给定一个由函数 p_1，…，p_n 组成的程序 P，P 的函数调用图是由节点集合 $N=\{p_1,\ldots,p_n\}$、边集合 $E\subseteq N\times N$，以及一个区别对待的入口节点 $r\in N$ (代表主函数)组成的图 $G_p=<N,E,r>$ (通常就写作 G)，其中对于每一个 $e_k=<p_i,p_j>$，k 的权值表示 p_i 中对 p_j 的调用点个数。如果 p_i 对 p 有 n（n>1）个调用点，则连接 p_i 和 p 的边的权值为 n。

作为调用图的一个例子，考虑图 4-3 中的程序框架。过程_main 调用 sub_401070

和 sub_40112b，sub_401070 调用 sub_4010A0 和 sub_4010C0，sub_4010C0 分别调用 sub_40112B 和_strcpy；因为_main 有两次对 sub_401070 的调用，所以连接_main 与 sub_401070 的边的权值为 2。图 4-4 给出这个程序的函数调用图。

```
.text:00401000 _main            proc near
.text:00401021                  call sub_401070
.text:00401030                  call sub_401070
.text:0040105B                  call sub_40112B
.text:00401063 _main            endp

.text:00401070 sub_401070       proc near
.text:00401077                  call sub_4010A0
.text:00401092                  call sub_4010C0
.text:0040109B sub_401070       endp

.text:004010A0 sub_4010A0       proc near
.text:004010A7                  call sub_4010C0
.text:004010B0 sub_4010A0       endp

.text:004010C0 sub_4010C0       proc near
.text:004010D6                  call _strcpy
.text:004010F1                  call sub_40112B
.text:004010F9 sub_4010C0       endp

.text:0040112B sub_40112B       proc near
.text:00401134 sub_40112B       endp
```

图4-3 示例程序段

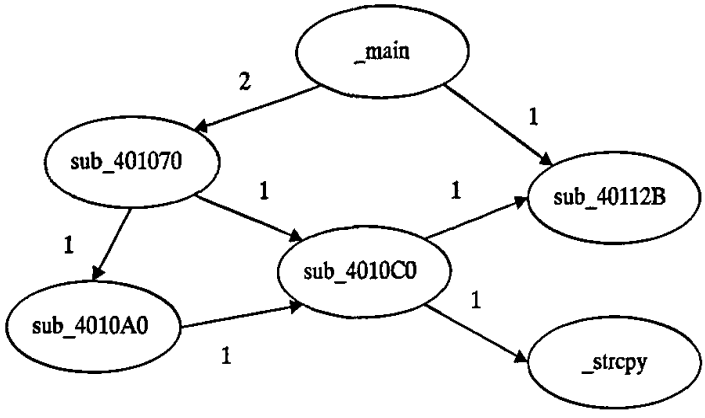

图4-4 函数调用图

4.5 静态分析模块实现

该静态分析模块是基于上节描述的函数调用图进行的。从图 4-4 可以看出，静态分析可以采用两种分析顺序：一是自顶向下的顺序，该顺序需要处理调用图中存在的每一条路径，因此分析量很大；二是自底向上的顺序。因为我们的分析工具的根本出发点是基于 C 库和操作系统中的不安全函数调用，如果将调用图看作一棵有向树，只对该有向树的叶子节点中像 strcpy 这样的不安全函数的节点做自底向上的分析，则可以避免搜索调用图的所有路径，减小分析量。例如对于图 4-4，采用自底向上的分析顺序，只需要从叶子节点_strcpy 开始搜索，搜索的路径只有 4 条。

下面以对 strcpy 函数调用的处理为例，详细介绍静态分析的算法。

首先给出三个相关的函数声明，这三个函数都是作为 4.2 节列出的 CProcUnit 类的成员函数实现。

1. void CProcUnit::StrcpyHandler(CALLEDPROC &cp);

   功能：

   strcpy 处理函数，如果某个参数是调用函数的形参，则调用 StrcpyRecursionHandler 回溯到该参数的最终来源

   参数：

   cp：包含 strcpy 函数调用的实参等信息

2. void CProcUnit::StrcpyOverflowJudge(vector<ARGUS> &vArgus,

vector<REFINFO> &vRefInfo);

功能：

判断 strcpy 函数调用是否导致缓冲区溢出

参数：

vArgus： strcpy 函数调用的实参，按函数调用的压栈顺序保存

vRefInfo：函数调用关系信息

3. void CProcUnit::StrcpyRecursionHandler(CProcUnit *pRefProcUnit,
   CProcUnit *pRefedProcUnit, vector<ARGUS> vArgus,
   vector<REFINFO> vRefInfo, vector<string> vRecursion);

功能：

求 strcpy 实参信息的递归函数，当普通被调用函数的实参是调用函数的形参时，递归求出最终的实参信息

参数：

pRefProcUnit：调用函数

pRefedProcUnit：被调用函数

vArgus：strcpy 的实参信息

vRefInfo：保存函数调用关系，输出漏洞信息时使用

vRecursion：保存函数调用关系，避免递归调用时陷入死循环

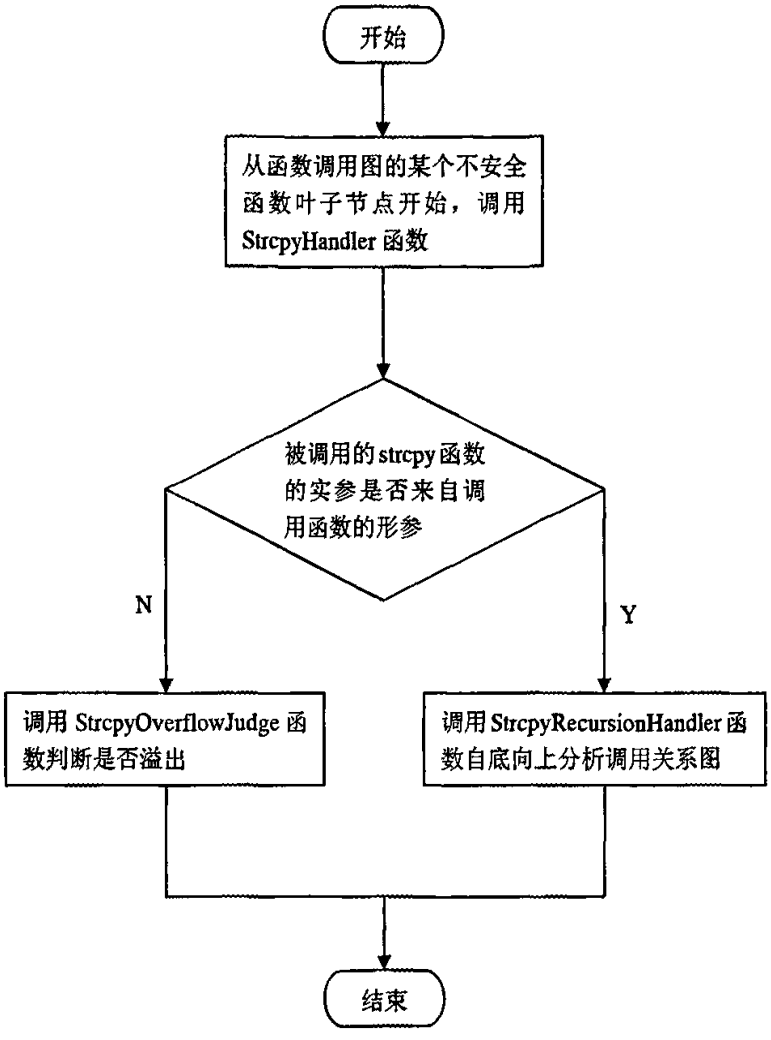

图4-5 静态分析算法流程

图 4-5 显示了静态分析算法的流程。该算法从函数调用图的某个不安全函数叶子节点开始，调用相应的处理函数。StrcpyHandler 对应于不安全函数 strcpy 的处理函数。因此，对于其它不安全函数，也可以添加相应的处理函数，例如对于 scanf，可以添加一个相应的 ScanfHandler 函数。如果在该调用点被调用的不安全函数的所有实参都不是调用函数的形参，则此时可以直接根据实参的信息，调用溢出判断函数作出是否导致栈溢出的判断。否则，如果此时有实参来自调用函数的形参，则需要调用相应的*RecursionHandler 函数自底向上分析调用关系图，更新最终的实参信息才能作出是否导致溢出的判断。以图 4-4 的函数调用图为例，函数

sub_4010C0 调用不安全函数_strcpy。_strcpy 接受两个参数，目的缓冲区 des 和源缓冲区 src。如果参数 des 和 src 均是调用函数 sub_4010C0 的局部堆栈变量，则通过代码分析模块的处理，可以直接求出实参的缓冲区范围，从而判断是否会导致栈缓冲溢出。如果参数 des 或 src 来自调用函数 sub_4010C0 的形参，则此时 des 或 src 的缓冲区范围只能通过函数调用关系才能求出。所以，需要调用 StrcpyRecursionHandler 分析函数的调用关系。

图 4-6 显示了 StrcpyRecursionHandler 的流程。同样，对于其它的不安全函数调用，也可以添加相应的*RecursionHandler 处理函数。该函数遍历与该节点相关的每个调用点，根据调用点处的调用信息，用调用函数的实参信息更新被调用函数的形参信息。如果在该调用点处还存在参数传递关系，则通过递归调用相应的*RecursionHandler 函数，继续向上分析函数的调用关系，直到所有的实参信息都得到了最终的更新。以 4-4 的调用图为例，如果_strcpy 的某个参数是 sub_4010C0 的形参，则调用 StrcpyRecursionHandler 分析调用关系，更新实参信息。如果 sub_4010C0 对_strcpy 调用的实参来自 sub_4010C0 的形参，从调用图可以看出，sub_4010C0 的被调用点有两个，所以这里需要分 sub_4010A0 和 sub_401070 两路继续调用 StrcpyRecursionHandler，直到实参信息得到了最终的更新。

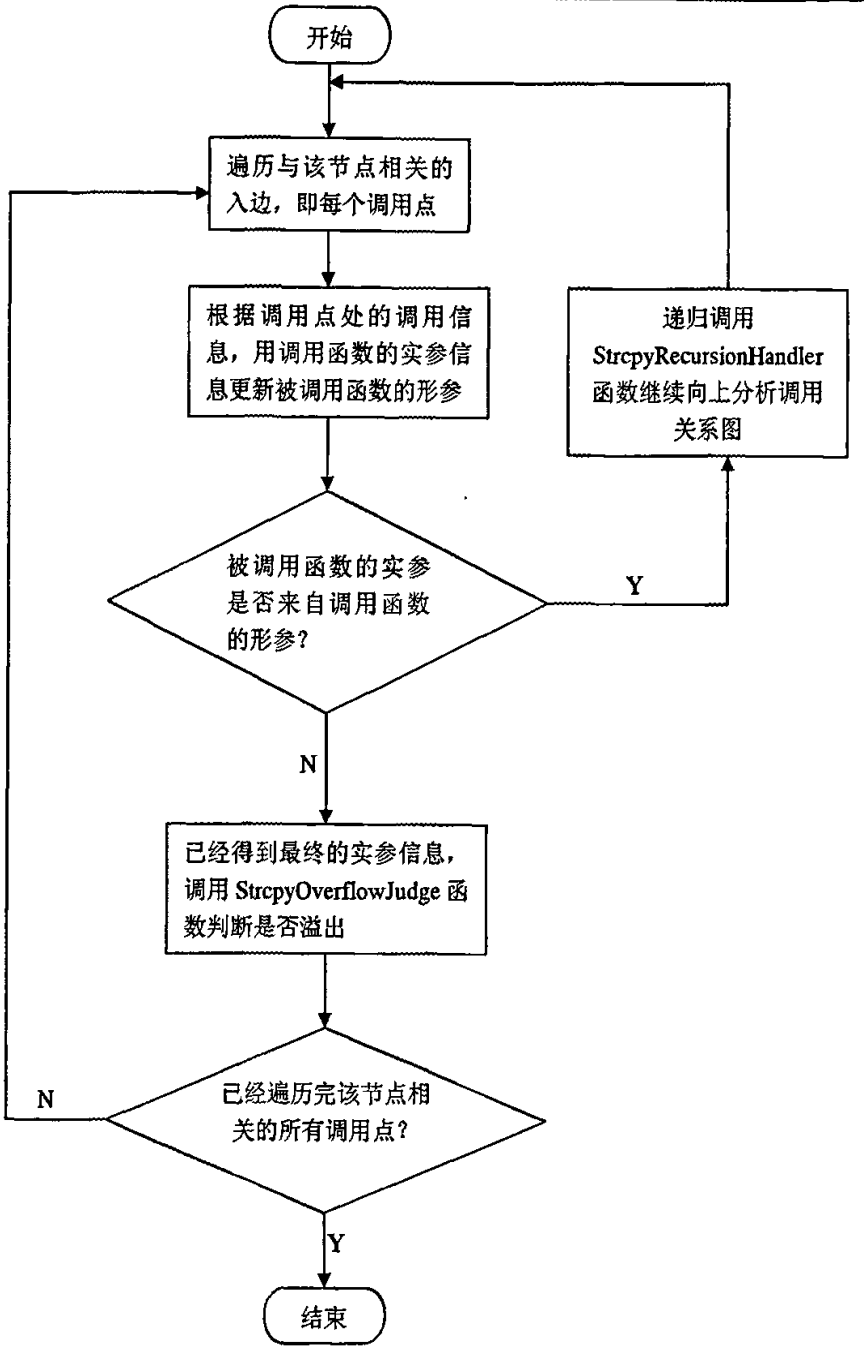

图4-6 StrcpyRecursionHandler 函数流程

4.6 小结

在第二章和第三章的相关知识介绍的基础上，本章主要对针对二进制代码的栈缓冲溢出检测原型系统的设计与实现作了详尽的描述，主要包括：系统总体结构、代码解析模块实现、静态分析模块实现。整个原型系统通过功能模块划分独立完成，最终集成为一个栈缓冲溢出检测系统。下一章将介绍对该原型系统的测试。

第五章　测试

5.1　测试目的

目前实现的该原型系统还不完善，只考虑了 C 语言中与字符串操作函数相关的数据类型，包括字符数组、字符串指针和字符串指针数组等在反汇编代码中的处理，没有对比较复杂的数据结构，例如结构体等的反汇编代码做分析。因此，下一节的测试实例是基于下述目的所设计的：

1. 测试在堆栈中分配的字符数组、字符串指针；
2. 测试字符串指针的赋值操作；
3. 测试字符串拷贝函数 strcpy；
4. 测试跨函数的调用字符串拷贝函数。

5.2　测试实例

下面的图 5-1 和图 5-2 是我们的测试源代码及对应的反汇编代码。代码左边的数字标识了代码在文件中的位置。

测试代码中定义了两个局部字符数组变量 buff600 和 buff100，分别是 600 字节和 100 字节大小的栈缓冲区；定义了两个局部字符串指针变量 p500 和 p100，其中 p500 对字符数组变量 buff600 作了加 100 的操作，因此 p500 指向一个 500 字节大小的栈缓冲区，不同的是 p100 对 buff600 作了加 500 的操作，因此 p100 指向一个 100 字节大小的栈缓冲区；主函数_tmain 直接调用了字符串拷贝函数 strcpy 四次，分别测试不同的缓冲区类型和缓冲区大小；定义了函数 fun，用于测试通过参数传递字符数组和字符串指针。

```
1. int _tmain(int argc, _TCHAR* argv[])
2. {
3.    char buff600[600];
4.    char buff100[100];
5.
6.    char *p500 = buff600 + 100;
7.    char *p100 = buff600 + 500;
8.
9.    //局部关系
10.   strcpy(buff600, buff100);
11.   strcpy(buff100, buff600);
12.   strcpy(buff100, p500);
13.   strcpy(buff100, p100);
14.   //参数传递关系
15.   fun(buff100);
16.   fun(buff600);
17.   fun(p100);
18.   return 0;
19. }
```

```
20. void fun(char*buff)
21. {
22.   char buff600[600];
23.   char buff100[100];
24.
25.   char *p500 = buff600 + 100;
26.   char *p100 = buff600 + 500;
27.
28.   strcpy(buff600, buff);
29.   strcpy(p500, buff);
30.   strcpy(buff100, buff);
31.   strcpy(p100, buff);
32.
33.   strcpy(buff, buff600);
34.   strcpy(buff, buff100);
35.
36.   strcpy(buff, p500);
37.   strcpy(buff, p100);
38. }
```

图5-1 测试源代码

```
2. .text:00401000 sub_401000 proc near
   ......
27. .text:0040102E    call    _strcpy
   ......
33. .text:0040103E    call    _strcpy
   ......
39. .text:0040104E    call    _strcpy
   ......
45. .text:0040105E    call    _strcpy
   ......
51. .text:00401071    call    _strcpy
   ......
57. .text:00401081    call    _strcpy
   ......
63. .text:00401091    call    _strcpy
   ......
69. .text:004010A1    call    _strcpy
   ......
76. .text:004010B4 sub_401000 endp
```

```
80. .text:004010C0 _main      proc near
   ......
106. .text:004010EE    call    _strcpy
   ......
112. .text:00401101    call    _strcpy
   ......
118. .text:00401111    call    _strcpy
   ......
124. .text:00401121    call    _strcpy
   ......
128. .text:0040112D    call sub_401000
   ......
132. .text:0040113C    call sub_401000
   ......
136. .text:00401148    call sub_401000
   ......
145. .text:0040115D _main       endp
```

图5-2 测试反汇编代码

5.3 测试结果

图 5-3 以树的形式显示了函数的调用关系。从图中可以看出，函数_main 对 sub_401000 有三个调用点，对应于源代码中函数_tmain 对函数 fun 的三次调用。

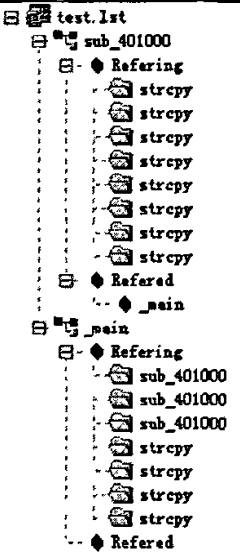

图5-3 测试结果：函数调用关系

图 5-4 显示了部分扫描的结果。从图 5-4 可以看出，扫描程序通过计算字符串拷贝函数所操作的目的和源缓冲区相对返回地址的偏移即最大值，和与相邻缓冲区的偏移即最小值，对是否溢出做出判断。扫描结果显示：

1. 发现反汇编代码中 112 行，对应于源程序中 11 行的由于局部变量的拷贝引起的缓冲溢出，目标缓冲区范围[104，116]，源缓冲区范围[604，724]。
2. 发现反汇编代码中 118 行，对应于源程序中的 12 行由于局部变量的拷贝引起的缓冲区溢出，目标缓冲区范围[104，116]，源缓冲区范围[504，624]。
3. 发现反汇编代码中由于函数调用_main(80 行) →sub_401000(132 行) →_strcpy(39 行)引起缓冲区溢出，目标缓冲区范围[104，116]，源缓冲区范围[604，724]。对应于源代码中的函数调用_tmain→fun(16 行) →strcpy(30 行)。
4. 发现反汇编代码中由于函数调用_main(80 行) →sub_401000(132 行) →_strcpy(45 行)引起缓冲区溢出，目标缓冲区范围[104，224]，源缓冲区范围[604，724]。对应于源代码中的函数调用_tmain→fun(16 行) →strcpy(31 行)。
5. 发现反汇编代码中由于函数调用_main(80 行) →sub_401000(128 行) →_strcpy(51 行)引起缓冲区溢出，目标缓冲区范围[104，116]，源缓冲区范围[604，724]。对应于源代码中的函数调用_tmain→fun(15 行) →strcpy(33 行)。
6. 发现反汇编代码中由于函数调用_main(80 行) →sub_401000(136 行) →_strcpy(51 行)引起缓冲区溢出，目标缓冲区范围[104，224]，源缓冲区范围[604，

724]。对应于源代码中的函数调用_tmain→fun(17 行) →strcpy(33 行)。

7. 发现反汇编代码中由于函数调用_main(80 行) →sub_401000(128 行) →_strcpy(63 行)引起缓冲区溢出，目标缓冲区范围[104，116]，源缓冲区范围[504，624]。对应于源代码中的函数调用_tmain→fun(15 行) →strcpy(36 行)。
8. 发现反汇编代码中由于函数调用_main(80 行) →sub_401000(136 行) →_strcpy(63 行)引起缓冲区溢出，目标缓冲区范围[104，224]，源缓冲区范围[504，624]。对应于源代码中的函数调用_tmain→fun(17 行) →strcpy(36 行)。

扫描结果表明，该扫描工具能够检测出字符数组、字符串指针的不安全拷贝操作导致的栈缓冲溢出。

函数	行号	结果	参数1	参数1	参数2	参数2
函数调用关系			最大值	最小值	最大值	最小值
_main	80					
strcpy	106	安全	724	604	116	104
_main	80					
strcpy	112	溢出	116	104	724	604
_main	80					
strcpy	118	溢出	116	104	624	504
_main	80					
strcpy	124	可能溢出	116	104	224	104
_main	80					
sub_401000	128					
strcpy	27	安全	724	604	116	104
_main	80					
sub_401000	132					
strcpy	27	可能溢出	724	604	724	604
_main	80					
sub_401000	136					
strcpy	27	安全	724	604	224	104
_main	80					
sub_401000	128					
strcpy	33	安全	624	504	116	104
_main	80					
sub_401000	132					
strcpy	33	可能溢出	624	504	724	604
_main	80					
sub_401000	136					
strcpy	33	安全	624	504	224	104
_main	80					
sub_401000	128					
strcpy	39	可能溢出	116	104	116	104
_main	80					
sub_401000	132					
strcpy	39	溢出	116	104	724	604
_main	80					
sub_401000	136					
strcpy	39	可能溢出	116	104	224	104
_main	80					
sub_401000	128					
strcpy	45	可能溢出	224	104	116	104
_main	80					
sub_401000	132					
strcpy	45	溢出	224	104	724	604

图5-4 测试扫描结果

第六章 结束语

经过十几年的发展，缓冲区溢出已经成为一种成熟和最有效的黑客攻击手段。并且在接下来的许多年里，情况仍然会是这样。为了提高系统中软件的安全性，进行缓冲区溢出发现技术的研究具有非常重要的意义。目前已经有多种方法用于检测缓冲区溢出，包括静态的和动态的。基于源码的静态检测技术是当前静态检测的主流技术。静态检测技术具有高效快速的特点，只需要很短的时间即可完成对项目源代码的检测，并且检查者不需要了解程序的实现方式。但是目前比较成型的静态检测工具所采用的方法有很多缺点：都只做到了词法分析，然后根据构造的数据库，通过匹配的方法判断是否符合溢出模式；不够准确，很多错误检测不到，或者检测到的并非错误。

本文借鉴了参与开发的基于源代码的静态检测工具所采用的技术，在对二进制代码文件的格式分析、汇编级的指令还原、汇编指令级的漏洞语法研究的基础之上，从形式描述的角度研究缓冲区溢出的检测问题。

首先对反汇编的二进制文件进行分析，提取与栈缓冲漏洞相关的特征和语法模式并将基于 C/C++源码的静态分析技术向汇编级的静态分析进行移植和扩展，然后设计了一个结构合理、性能良好的语法分析引擎进行与栈缓冲溢出相关语法成分的分析，最后以可能导致栈缓冲溢出的 C 库函数和 Windows 操作系统中类似的函数为出发点，进行是否导致栈溢出的分析。通过测试证明，基于本文的检测方法构造的原型系统能够检测出一些栈缓冲溢出的情况。

本文对非常复杂的数据结构没有分析，而且由于程序分析问题的复杂性，在实例实现上只是给出了几种简单的情况检测。因此，从描述方法的完备性和实现方法的完全性两个方面，都还有很多工作要做。这也是本文今后要进行改进的地方。

如果能够十分准确的检测出程序中的缺陷，保证错误全都检测出来，检测出来的都没有错误，那么无疑将会从根本上消除利用缓冲溢出漏洞进行攻击的可能性。事实上，由于静态检测方法本身的局限性，无法 100%的检测出程序的缓冲溢出漏洞，而如果采用动态方法，又有性能等方面的缺点，因此，如何将静态检测和动态检测方法结合起来也是一个改进和扩展的方向。

致谢

研究生的三年学习生活即将结束，在毕业论文完成之际，谨向在我攻读硕士学位期间指导、关心、帮助我的导师、同学和朋友们表示诚挚的感谢！

衷心的感谢导师李毅教授在我攻读硕士期间对我的热情关怀、鼓励和培养。自从我研究生入学以来，便始终得到李老师的悉心指导和亲切关怀。李老师不仅在学术研究上对我严格要求、悉心指导，而且在立身做人方面言传身教，生活上对我关心爱护，使我不仅在学习上得到长足的进步，而且在个人修养方面也受益匪浅。从李老师身上，我不仅学到了知识，更重要的是，我还明白了许多做人的道理，这将使我受益终生。

感谢我的家人和所有关爱我的朋友们，谢谢你们给予我精神上的鼓励理解，和物质上的支持，让我能顺利完成研究生学业！

参考文献

[1] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings Volume 2, 5-27 Jan. 2000, pages 119 - 129 vol.2.

[2] CERT/CC. http://www.cert.org/stats.

[3] 绿盟科技. http://www.nsfocus.net.

[4] 鄢喜爱, 杨金民. 缓冲区溢出攻击的分析与防范. 网络安全技术与应用, 2006 年, 03 期.

[5] D. Pozza, R. Sisto, L. Durante, and A. Valenzano. Comparing Lexical Analysis Tools for Buffer Overflow Detection in Network Software. Communication System Software and Middleware, 2006. Comsware 2006. First International Conference on 08-12 Jan. 2006, pages 1-7.

[6] J. Heffley, P. Meunier. Can source code auditing software identify common vulnerabilities and be used to evaluate software security? System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on 5-8 Jan. 2004, page 10.

[7] Aishwarya Iyer, L.M. Liebrock. Vulnerability scanning for buffer overflow. Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on Volume 2, 2004 pages 116 - 117 Vol.2.

[8] Tzi-Cker Chiueh, Fu-Hau Hsu. RAD: a compile-time solution to buffer overflow attacks. Distributed Computing Systems, 2001. 21st International Conference on 16-19 April 2001 pages 409 – 417.

[9] J. Pincus, B. Baker. Beyond stack smashing: recent advances in exploiting buffer overruns. Security & Privacy Magazine, IEEE. Volume 2, Issue 4, Jul-Aug 2004 pages 20-27.

[10] M. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). Computer Security Applications Conference, 2004. 20th Annual 6-10 Dec. 2004 pages 82-90.

[11] D. Wagner, J. Foster, E. Brewerand, A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In Proceedings of the Year 2000 Network and Distributed System Security Symposium (NDSS), pages 3-17, San Diego, CA, 2000.

[12] Yu-an Tan, Ji-yan Zheng, Yuan-da Cao, Xue-lan Zhang. Buffer overflow protection based on adjusting code segment limit. Communications and Information Technology, 2005. ISCIT 2005.

IEEE International Symposium on Volume 2, 12-14 Oct 2005 pages 947-950.

[13] Zhenkai Liang, R. Sekar. Automatic generation of buffer overflow attack signatures: an approach based on program behavior models. Computer Security Applications Conference, 21st Annual 5-9 Dec. 2005 page 10 pp.

[14] Flawfinder. http://www.dwheeler.com/flawfinder/.

[15] RATS. http://www.fortifysoftware.com/security-resources/rats.jsp.

[16] John Viega, J.T. Bloch, Tadayoshi Kohno, Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. Reliable Software Technologies, 2000.

[17] Expat. http://expat.sourceforge.net/.

[18] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, David Vitek. Buffer Overrun Detecting using Linear Programming and Static Analysis. CCS'03, Washington, DC, USA. October 2003, pages 27-30.

[19] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle and QianZhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In proceedings of the 7th USENIX Security Symposium, pp. 63-77, San Antonio, TX, Jan. 1998. USENIX.

[20] StackShield. http://www.angelfire.com/sk/stackshield.

[21] P. Staff. Bypassing StackGuard and StackShield. Phrack Magazine, 5(56), 2000.

[22] San. Hacking Windows CE Phrack Magezine. Phrack Magazine, 06(63), 2005.

[23] ssp. http://www.research.ibm.com/trl/projects/security/ssp/.

[24] Kyung-suk Lhee, Steve J. Chapin. Type-Assisted Dynamic Buffer Overflow Detection. USENIX Security Symposium 2002: 81-88.

[25] A. Baratloo, N. Singh, T. Tsai. Transparent run-time defense against stack smashing attacks (libsafe-libverify). In Proceedings of 2000 USENIX Annual Technical Conference, San Diego, California, USA, 2000:18-23.

[26] PC-lint. http://www.gimpel.com/html/pcl.htm.

[27] David Larochelle, David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In 2001 USENIX Security Symposium, Washington, D.C., August 12-17, 2001.

[28] K. Piromsopa, R.J. Enbody. Buffer-Overflow Protection: The Theory. Electro/information Technology, 2006 IEEE International Conference on May 2006 pages 454 – 458.

[29] Alexander Aiken. Set constraints: results, applications, and future directions. In PPCP'94: Principles and Practice of Constraint Programming. Springer-Verlag, 1994.

[30] M. Zitser, R. Lippmann, T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. SIGSOFT Softw.Eng.Notes, 29(6):97-106, 2004.

[31] M. Hind, M. Burke, P. Carini, J.D.Choi. Interprocedural pointer alias analysis. TOPLAS, 21(4): 848-894, 1999.

[32] M. Das. Unification-based pointer analysis with directional assignments. In PLDI, 2000.

[33] N. Heintze, O. Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In PLDI, 2001.

[34] Chris Lattner, Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, 2nd IEEE/ACM International Symposium on Code Generation and Optimization, 20-24 March 2004, CA, USA.

[35] CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. ACM Conf. on PLDI'03, 2003, June 9-11, 2003, San Diego, California, USA.

[36] Yiche Xie, Andy Chou, Dawson Engler. ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. ESEC/FSE'03, September 1-5, 2003, Helsinki Finland.

[37] Michael Howard, David Leblanc. 编写安全的代码 (程永敬等译). 北京: 机械工业出版社, 2005.1.

[38] Shankar U., Talwar K., Foster J.S. Detecting format string vulnerabilities with type qualifiers. USENIX Security Symposium, USA, 2001.

[39] 肖道举, 陈博文, 陈晓苏. 一种基于程序逻辑结构分析的缓冲区溢出攻击抵御方法. 计算机工程与科学, 2005 年, 05 期.

[40] 张明军, 罗军. 缓冲区溢出静态分析中的指针分析算法. 计算机工程, 2005 年, 18 期.

[41] 邱晓鹏, 张玉清, 冯登国. 一种防止堆缓冲区溢出的新方法. 中国科学院研究生院学报, 2005 年, 02 期.

[42] 赵天福, 王海生. 基于可执行串搜索的缓冲区溢出检测技术研究. 高性能计算技术, 2006 年, 02 期.

[43] 崔志刚, 谭毓安, 曹元大, 张雪兰. 基于段长限制的缓冲区溢出保护. 计算机工程, 2006 年, 10 期.

[44] 朱虹, 丁雁林. 缓冲区溢出检测模型研究. 计算机工程与应用, 2006 年, 10 期.

[45] 唐福华, 朱鸿宇, 刘瑰. 一种发现 C 程序中缓冲区溢出漏洞的算法. 计算机应用与软件, 2006 年, 07 期.

[46] 刘应华, 姜建国. 缓冲区溢出建模分析. 计算机工程与应用, 2005 年, 03 期.

[47] 黄金志, 胡健生, 廖赟, 柴仁文. 基于 Petri 网的程序缓冲区溢出检测方法. 计算机应用,

2005 年, 05 期.

[48] 高仲仪, 金茂忠. 编译原理及编译程序构造. 北京航空航天大学出版社, 1990.12.

[49] Kenneth C. Louden. 编译原理及实践(冯博琴等译). 北京: 机械工业出版社, 2000.

[50] 蒋卫华, 李伟华, 杜君. 缓冲区溢出攻击:原理,防御及检测. 计算机工程, 2003 年, 10 期.

[51] 朱明, 尹大成, 陈亿霖. 缓冲区溢出攻击及检测方法研究. 计算机工程, 2002 年, 07 期.

个人简历及攻读硕士学位期间的研究成果

向文杰，男，1982 年 4 月 25 日出生。

2004 年 6 月毕业于北京师范大学计算机科学与技术专业。

2004 年 9 月开始在电子科技大学计算机科学与工程学院攻读计算机应用专业硕士学位。

参加的科研项目：

1. 2005 年 7 月－2006 年 5 月 计算机学院东 210 实验室
   四川省安全厅信息安全项目
2. 2006 年 5 月－2007 年 1 月 计算机学院东 210 实验室
   东方电气集团《DEA 组态软件》项目

攻读硕士学位期间发表的论文：

[1] 向文杰, 李毅. Linux 环境下 ELF 文件型病毒的分析. 科技信息, 2006.9.